

En tu AMSTRAD CPC



"Una guía para programadores de 8 bit en el siglo XXI"

Jose Javier García Aranda

INDICE

1 ¿Por qué programar en 2016 una maquina de 1984?

Porque las limitaciones no son un problema sino una fuente de inspiración.

Las limitaciones, ya sean de una maquina o de un ser humano, o en general de cualquier recurso disponible estimulan nuestra imaginación para poder superarlas. El AMSTRAD, una maquina de 1984 basada en el microprocesador Z80, posee una reducida memoria (64KB) y una reducida capacidad de procesamiento, aunque sólo si lo comparamos con los ordenadores actuales. Esta máquina es en realidad un millón de veces más rápida que la que construyó Alan Turing para descifrar los mensajes de la maquina enigma en 1944

Como todos los ordenadores de los años 80, el AMSTRAD CPC arrancaba en menos de un segundo, con el intérprete BASIC dispuesto a recibir comandos de usuario, siendo el BASIC el lenguaje con el que los programadores aprendían y hacían sus primeros desarrollos. El BASIC del AMSTRAD era particularmente rápido en comparación al de sus competidores. Y estéticamente muy atractivo!



Fig. 1. El mítico AMSTRAD modelo CPC464

En cuanto al microprocesador Z80 ni siquiera es capaz de multiplicar (en BASIC puedes multiplicar pero eso se basa en un programa interno que implementa la multiplicación mediante sumas o desplazamientos de registros), tan solo puede hacer sumas, restas y operaciones lógicas. A pesar de ello era la mejor CPU de 8 bit y tan sólo constaba de 8500 transistores, a diferencia de otros procesadores como el M68000 cuyo nombre precisamente le viene de tener 68000 transistores

CPU	Número de transistores	MIPS (millones de instrucciones por segundo)	Ordenadores y consolas que lo incorporan
6502	3.500	0.43 @1Mhz	COMMODORE 64, NES, ATARI 800...
Z80	8.500	0.58 @4Mhz	AMSTRAD, COLECOVISION, SPECTRUM, MSX...
Motorola 68000	68.000	2.188 @ 12.5 Mhz	AMIGA, SINCLAIR QL, ATARI ST...
Intel 386DX	275.000	2.1 @16Mhz	PC
Intel 486DX	1.180.000	11 @ 33 Mhz	PC
Pentium	3.100.000	188 @ 100Mhz	PC
ARM1176		4744 @ 1Ghz (1186 por core)	Raspberry pi 2, nintendo 3DS, samsung galaxy,…
Intel i7	2.600.000.000	238310 @ 3Ghz (casi 500.000 veces mas rápido que un Z80 !)	PC

Tabla 1Comparativa de MIPS

Ello hace que programarlo sea extremadamente interesante y estimulante para lograr resultados satisfactorios. Toda nuestra programación debe ir orientada a reducir complejidad computacional espacial (memoria) y temporal (operaciones), obligándonos a inventar trucos, artimañas, algoritmos, etc, y haciendo de la programación una aventura apasionante. Es por ello, que la programación de máquinas de baja capacidad de procesamiento es un concepto atemporal, no sujeto a modas ni condicionado por la evolución de la tecnología.

Todo el código de este libro, incluida la librería para que hagas tus propios juegos o para que hagas contribuciones a la librería , lo puedes en contrar en el proyecto GitHub “8BP”, en esta URL:

https://github.com/jjaranda13/8BP

2 Funciones de 8BP y uso de la memoria

La librería 8BP no es un "motor de juegos". Es algo intermedio entre una simple extensión de comandos BASIC y un motor de juegos.
Los motores de juegos como el game maker, el AGD (arcade game designer), el unity, y muchos otros, limitan en cierta medida la imaginación del programador, obligándole a usar unas determinadas estructuras, a programar en lenguaje limitado de script la lógica de un enemigo, a definir y enlazar pantallas de juego, etc

La librería 8BP es diferente. Es una librería capaz de ejecutar deprisa aquello que el BASIC no puede hacer. Cosas como imprimir sprites a toda velocidad, o mover bancos de estrellas por la pantalla, son cosas que el BASIC no puede hacer y 8BP lo consigue

BASIC es un lenguaje interpretado. Eso significa que cada vez que el ordenador ejecuta una línea de programa debe primero verificar que se trata de un comando válido, comparando la cadena de caracteres del comando con todas las cadenas de comandos validos. A continuación debe validar sintácticamente la expresión, los parámetros del comando e incluso los rangos permitidos para los valores de dichos parámetros. Además los parámetros los lee en formato texto (ASCII) y debe convertirlos a datos numéricos. Finalizada toda esta labor, procede con la ejecución. Pues bien, toda esa labor que se realiza en cada instrucción es la que diferencia un programa compilado de un programa interpretado como los escritos en BASIC.

Dotando al BASIC de los comandos proporcionados por 8BP, es posible hacer juegos de calidad profesional, ya que la lógica del juego que programes puede ejecutarse en BASIC mientras que las operaciones intensivas en el uso de CPU como imprimir en pantalla o detectar colisiones entre sprites, etc son llevadas a cabo en código máquina por la libería.

Sin embargo, no todo es facilidad y ausencia de problemas. Aunque la librería 8BP te va a proporcionar funciones muy útiles en videojuegos, deberás usarla con cautela pues cada comando que invoques atravesará la capa de análisis sintáctico del BASIC, antes de llegar al inframundo del código máquina donde se encuentra la función, por lo que el rendimiento nunca será el óptimo. Deberás ser astuto y ahorrar instrucciones, medir los tiempos de ejecución de instrucciones y trozos de tu programa y pensar estrategias para ahorrar tiempo de ejecución. Toda una aventura de ingenio y diversión. Aquí aprenderás como hacerlo e incluso te presentaré una técnica a la que he llamado "lógicas masivas" que te permitirá acelerar tus juegos a límites que quizás considerabas imposibles.

Además de la librería, tienes a tu disposición un sencillo pero completo editor de sprites y gráficos y una serie de herramientas magníficas que te permitirán disfrutar en el siglo XXI de la aventura de programar un microordenador.

2.1 Funciones de 8BP

Tras cargar la librería con el comando: LOAD "8BP.BIN" e invocar desde BASIC la función _INSTALL_RSX (definida en código máquina) mediante el comando BASIC:

CALL &6b78

Dispondrás de los siguientes comandos, que aprenderás a usar con este libro (fíjate que aparece una barra vertical al principio de cada uno por ser "extensiones" del BASIC):

|PRINTSP, #,y,x	imprime un solo sprite (# es su numero) sin tener en cuenta byte de status
|PRINTSPALL, anima, sync	imprime todos los sprites con flag de impresion activo
|LOCATESP, #,y,x	cambia las coordenadas de un sprite (sin imprimirlo)
|POKE,dir,valor	introduce un dato 16bit (que puede ser negativo) en una dirección de memoria
|PEEK,dir,@valor%	lee un dato 16bit (que puede ser negativo) de una dirección
|COLAY, #,@colision	detecta la colision con el layout y retorna 1 si hay colision
|COLSP, #,@id	retorna primer sprite con el que colisiona #
|ANIMA, #	cambia el fotograma de un sprite segun su secuencia
|ANIMALL	cambia el fotograma de los sprites con flag animacion activado
|MOVER, #,dy,dx	movimiento relativo de un solo sprite
|MOVERALL, dy,dx	movimiento relativo de todos los sprites con flag de mov relativo activo
|AUTO, #,dy,dx	movimiento automatico de un sprite de acuerdo a su velocidad
|AUTOALL, dy,dx	movimiento de todos los sprites con flag de mov automatico activo
|STARS, bank,num,color,dy,dx	scroll de un banco de estrellas
|SETLIMITS, xmin,xmax,ymin,ymax	define la ventana de juego, donde se hace clippling
|LAYOUT, y,x,@string	imprime un layout de imagenes de 8x8 y rellena map layout
|SETUPSP, #, param_number, valor	modifica un parametro de un sprite
|SETUPSQ, #, adr0,adr1,...,adr7	crea una secuencia de animacion
|MUSIC, cancion,speed	comienza a sonar una melodía a la velocidad deseada
|MUSICOFF	deja de sonar la melodía

Tabla 2 Comandos disponibles en la librería 8BP

Adicionalmente dispones de un comando experimental:

|RETROTIME, fecha

Este comando permite transformar tu CPC en una maquina del tiempo, con solo introducir la fecha de destino deseada. La única limitación del comando es que debes introducir una fecha igual o posterior a la del nacimiento del AMSTRAD CPC, abril de 1984,

|RETROTIME, "01/04/1984"

Por favor, utiliza esta funcionalidad con precaución. Podrías crear una paradoja temporal y destruir el mundo.

Aunque de momento puedas tener cierto escepticismo respecto lo que puedes llegar a hacer con la librería 8BP, pronto descubrirás que el uso de esta librería junto con técnicas de programación avanzadas que aprenderás en este libro, te permitirá hacer juegos profesionales en BASIC, algo que quizás creías imposible.

2.2 Arquitectura del AMSTRAD CPC

Este apartado es útil para comprender posteriormente como usa la librería 8BP la memoria.

El amstrad es una computadora basada en el microprocesador Z80, funcionando a 4MHz. Como se aprecia en su diagrama de arquitectura, tanto la CPU como la matriz lógica de video (llamada "gate array") acceden a la memoria RAM, por lo que para poder "turnarse", los accesos a la memoria desde la CPU son retrasados, dando como resultado una velocidad efectiva de 3.3Mhz. Esto sigue siendo bastante potencia.

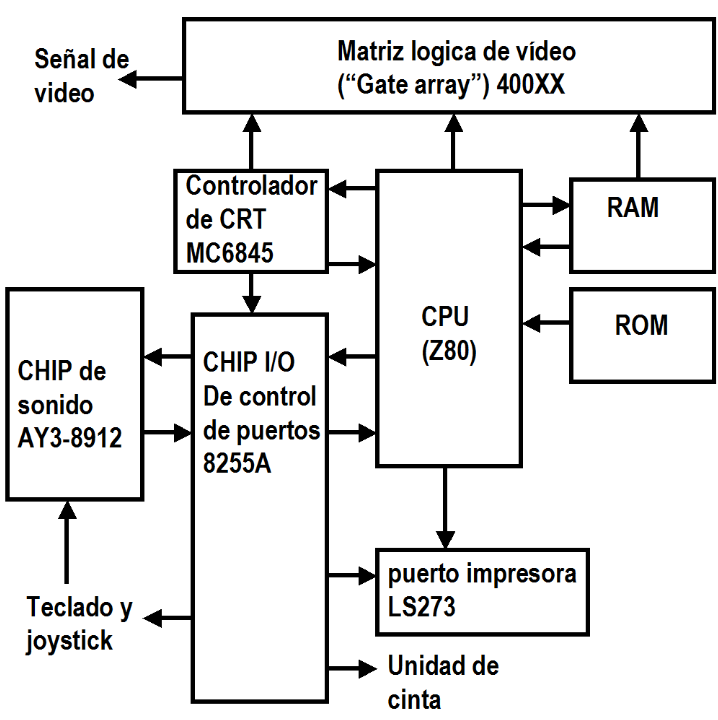


Fig. 2Arquitectura del AMSTRAD

La memoria de video, lo que vemos en la pantalla, es parte de las 64KB de memoria RAM, en concreto son las 16KB situadas en la zona superior de la memoria. La memoria se numera desde 0 hasta 65535 bytes. Pues bien, los 16KB comprendidos entre la dirección 49152 y 65535 es la memoria de video. En hexadecimal se representa como C000 hasta FFFF.

La memoria RAM de video es accedida por el gate array 50 veces por segundo para poder enviar una imagen a la pantalla. En ordenadores más antiguos (como el Sinclair ZX81) esta labor era encomendada al procesador, restándole aun más potencia.

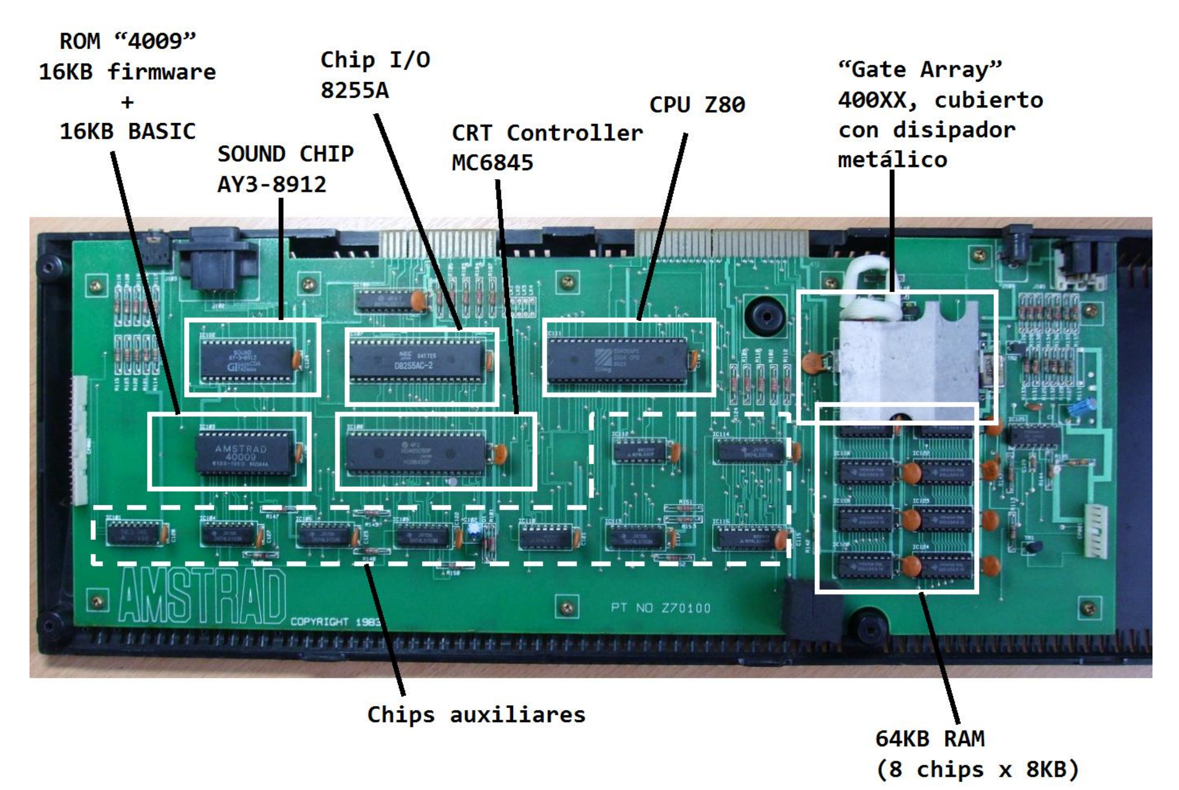


Fig. 3 Identificacion de componentes en la placa

El Z80 posee un bus de direcciones de 16bit, por lo que no es capaz de direccionar más de 64KB. Sin embargo el Amstrad posee 64kB RAM y 32kB ROM. Para poder direccionarlas, el AMSTRAD es capaz de "conmutar" entre unos bancos y otros, de modo que, por ejemplo si se invoca a un comando BASIC, se conmuta al banco de ROM donde se almacena el intérprete BASIC, que esta solapado con las 16KB de pantalla. Este mecanismo es sencillo y efectivo.

Además de la ROM que contiene el intérprete BASIC de 16KB situado en la zona de memoria alta, hay otras 16KB de ROM situadas en la memoria baja, donde se encuentran las rutinas del firmware (lo que podría considerarse el sistema operativo de esta máquina). En total (interprete BASIC y firmware) suman 32KB

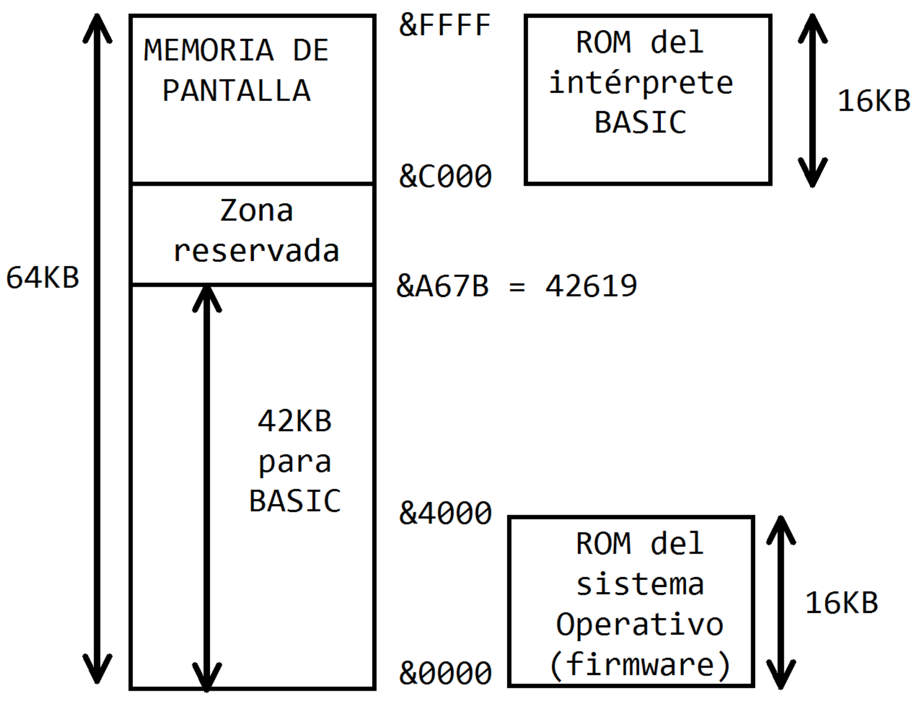


Fig. 4 Memoria del AMSTRAD

Como se aprecia en el mapa de memoria, de los 64KB de RAM, 16KB (desde &C000 hasta &FFFF) son la memoria de video. Los programas en BASIC pueden ocupar desde la posición &40 (dirección 64) hasta la 42619, pues mas allá hay variables del sistema. Es decir, que se dispone de unas 42KB para BASIC, tal y como podemos comprobar al imprimir la variable del sistema HIMEM (abreviatura de "High Memory").



Fig. 5 Variable de sistema HIMEM

El funcionamiento del BASIC tiene en cuenta el almacenamiento del programa en direcciones crecientes desde la posición &40 , mientras que una vez en ejecución, las variables que se declaran deben ocupar espacio para almacenar los valores que toman y puesto que no pueden ocupar la misma zona donde se almacena el programa, sencillamente se empiezan a almacenar en la dirección más alta posible, la 42619 y a medida que se usan más variables se consumen direcciones de memoria decrecientes.

En el AMSTRAD cada variable numérica de tipo entero ocupará 2 bytes de memoria.

2.3 Uso de la memoria de 8BP

La librería 8BP se carga en la zona de memoria alta disponible. Es importante entender como funciona el BASIC, para poder usar la librería.

El texto de un programa escrito en BASIC se almacena a partir de la dirección &40 pero una vez que empieza a ejecutarse, los valores que van tomando las variables de programa se almacenan ocupando posiciones decrecientes desde la 42619, de modo que si un programa es grande, podrían llegar a "chocar" con el propio texto del programa, destruyendo parte del mismo. Esto normalmente no va a ocurrir, no te preocupes.

La librería se carga a partir de la dirección 27000, destinando la memoria a las funciones, el player de musica, las canciones y los dibujos, tal y como se muestra en el siguiente diagrama

```
                       AMSTRAC CPC464 MAPA DE MEMORIA de 8BP

&FFFF +----------
      |  pantalla
&C000 +----------
      |  system (simbolos redefinibles, etc)
42619 +----------
      | banco 1 de 20 estrellas (desde 42580 hasta 42619)
42580 +----------
      | banco 0 de 20 estrellas (desde 42540 hasta 42579)
42540 +----------
      | map layout de caracteres (25x20 =500 bytes)
      |
42040 +----------
      | sprites (8KB para dibujos. 8040 bytes)
34000 +----------
      |  hasta 31 secuencias de animacion de 8 frames (496 bytes)
33500 +----------
      | canciones (1.5kB para musica)
      |
32000 +----------
      |  rutinas 8BP   ( y sobran 510 bytes)
      |     aqui estan todas las rutinas y la tabla de sprites
      |     incluye el player de musica "wyz"
27000 +----------
      |  | variables del basic
      |  V
      |
      |  ^ basic (texto del programa)
      |  |
      +---------- 0kB
```

Fig. 6 Memoria usando 8BP

Antes incluso de cargar la librería, deberás ejecutar el comando

MEMORY 26999

Para limitar el espacio ocupado por el BASIC. De esta manera, las variables de BASIC empezaran a ocupar espacio en ejecución desde la dirección 26999 hacia abajo. Tus programas pueden ser de casi 27KB de memoria, aunque como las variables ocupan espacio lo normal es que el tamaño máximo de tu programa sea inferior. No obstante hablamos de una cantidad de memoria muy respetable. Te costará mucho hacer un juego de 27KB, te lo aseguro.

3 Herramientas necesarias

Winape: emulador para S.O. windows con editor para editar y probar tu programa BASIC. Y tambien para ensamblar los gráficos y las músicas

Spedit: (“Simple Sprite Editor”) herramienta BASIC para editar tus graficos. El resultado de spedit es codigo en ensamblador que se envía a la impresora del amstrad CPC. Ejecutando la herramienta dentro de Winape, la impresora se redirige a un fichero de texto de modo que tus gráficos se almacenarán en un fichero txt. Esta herramienta ha sido creada para complementar a la librería 8BP

Wyztracker: para componer musica, bajo windows. El programa capaz de tocar las melodías compuestas por Wyztracker es el Wyzplayer, el cual está integrado dentro de 8BP. Tras ensamblar la música podrás hacerla sonar con un sencillo comando |MUSIC

Libreria 8BP: instala nuevos comandos accesibles desde BASIC para tu programa. Como comprobarás, esto va a ser el “corazón” que mueva la maquinaria que construyas.

CPCDiskXP : te permite grabar un disquete de 3.5” que luego podrás insertar en tu CPC6128 si dispones de un cable para conectar una disquetera. Si quieres hacer una cinta de audio para CPC464 esta herramienta no la necesitas

Opcionalmente:
fabacom: compilador ejecutable dentro del amstrad CPC 6128 o desde el emulador Winape para compilar tu programa BASIC y hacerlo ejecutar mas rápido. Es compatible con las llamadas a los comandos de la librería 8BP. Sin embargo no es recomendable por varios motivos:

- tu programa ocupará mucho mas pues fabacom necesita 10KB adicionales para sus librerías, y además, una vez que compila tu programa sigue ocupando lo mismo, de modo que un programa de 10KB se transforma en uno de 20KB.
- Hay documentados algunos problemas de incompatibilidad de este compilador con algunas instrucciones de BASIC.
- Además, como verás a lo largo de este libro, puedes lograr una velocidad muy alta sin necesidad de compilar.

4 Pasos que debes dar para hacer un juego

4.1 Tu juego en 5 ficheros

En este apartado vamos a ver los pasos que debes dar. No es algo secuencial, puedes ir haciendo gráficos a medida que programas e igual ocurre con las músicas. No te preocupes si ahora no entiendes con precisión cada paso. A lo largo del libro irás comprendiendo exactamente lo que significan con precisión. Y al final tienes un apéndice con información detallada a este respecto.

Lo que de momento debes entender es que tu juego se debe componer de 5 ficheros : 3 ficheros binarios y 2 ficheros BASIC

Los ficheros binarios son:

- Librería 8BP (es un fichero binario), incluyendo la tabla de atributos de sprites
- Fichero binario de musica con las melodías de tu juego
- Fichero binario de imágenes de sprites, incluyendo la tabla de secuencias de animacion

Y dos ficheros BASIC

- Cargador (carga la librería, musica y sprites y por último tu juego). Si además deseas hacer una pantalla de presentación que se muestre mientras se carga el juego, será lo primero que cargue este cargador
- Programa BASIC (tu juego)

Para hacer estos 5 ficheros debes dar estos pasos

PASO 1
Editar gráficos con SPEDIT
Ensamblar los gráficos con winape
Salvar los gráficos con el comando SAVE “sprites.bin”,b,33500, <tamaño>

PASO 2
Editar la música con WYZtracker
Ensamblar la música con winape. Las melodías se ensamblarán una detrás de otra, de modo que cada una comenzará en una dirección de memoria diferente que dependerá del tamaño que ocupen.
Salvar la música con el comando SAVE “music.bin”,b,32000, 1500

PASO 3
Re-ensamblar la librería 8BP, de modo que la parte de la librería que selecciona las melodías (el player wyz) pueda conocer en que direcciones de memoria se han ensamblado (hay mas dependencias pero esa es una de ellas). Una vez re-ensamblada, tendrás que salvarla con el comando
SAVE “8BP.LIB”, b,27000,5000

Esta será una versión de la librería especifica para tu juego. Por ejemplo el comando |MUSIC,3,5 hará sonar la melodía número 3 que tu mismo has compuesto. La melodía numero 3 puede ser completamente diferente en otro juego.

PASO 4
Cargar todo con un loader , que deberás hacer en BASIC. Por ejemplo:
10 MEMORY 26999
20 LOAD "!8bp.lib"
30 LOAD "!music.bin"
40 LOAD "!sprites.bin"
50 RUN "!tujuego.bas"

PASO 5
Programar tu juego, el cual debe primeramente ejecutar la llamada para instalar los comandos RSX, es decir CALL &6b78.
Tu juego lo puedes programar usando el editor de winape, mucho mas versátil que el editor del AMSTRAD y sirve tanto para editar ensamblador (.asm) como para editar BASIC (.bas)
Opcionalmente puedes compilar tu juego con fabacom y usar la versión compilada

PASO 6
Crear una cinta o un disco con tu juego

4.2 Crear un disco o una cinta con tu juego

4.2.1 Hacer un disco

Para crear un nuevo disco desde winape hacemos

File->drive A-> new blank disk

Con ello te aparecerá una ventana de administración de archivos para que le des nombre al nuevo fichero .dsk

Una vez creado ya puedes guardar ficheros con el comando SAVE. Para borrar un archivo se utiliza el comando “|ERA” (abreviatura de ERASE), que solo existe en CPC 6128 como parte del

|ERA,”juego.*”

y se borrarán

Para cargar el juego necesitas un cargador que cargue uno por uno los ficheros necesarios. Algo como:

```
10 MEMORY 26999
```

```
20 LOAD "!8bp.lib"
30 LOAD "!music.bin"
40 LOAD "!sprites.bin"
50 RUN "!mont7.bas"
```

Para salvar cada uno de los ficheros debes usar el comando SAVE con los parámetros necesarios, por ejemplo:

```
SAVE “LOADER.BAS”
SAVE “8BP.LIB”,b,27000,5000
SAVE “MUSIC.BIN”,b,32000,1500
SAVE “SPRITES.BIN”,b,33500,8500
SAVE “MONT7.BAS”
```

Si quieres grabar el .dsk en un disquete de 3.5” y conectarlo a una disquetera externa de tu AMSTRAD CPC 6128 , necesitarás el programa CPCDiskXP, muy sencillo de usar. A partir de un .dsk puede grabar un disquete de 3.5” en doble densidad (no olvides tapar el agujero del disquete para “engañar” al PC)

4.2.2 Hacer una cinta

Lo más importante al crear una cinta es guardar en ella los ficheros en el orden en el que van a ser cargados por el ordenador. Una cinta no es como un disco en el que puedes cargar cualquier fichero almacenado, sino que los ficheros se encuentran uno detrás de otro, por lo que debes poner especial cuidado en este punto.

Si tu cargador de juego es asi:

```
10 MEMORY 26999
20 LOAD "!8bp.lib"
30 LOAD "!music.bin"
40 LOAD "!sprites.bin"
50 RUN "!mont7.bas"
```

Entonces primero debes guardar el cargador (supongamos que se llama “loader.bas”), después el fichero “8BP.LIB”, despues “MUSIC.BIN”, después “SPRITES.BIN” y por último “MONT7.BAS”

Para crear un wav o un cdt desde winape

file->tape->press record

en ese momento te saldrá un menu de administración de archivos para que podamos decidir que nombre le damos al fichero wav o cdt

Si estas en modo CPC 6128, entonces a continuación debes ejecutar desde basic

|TAPE

y luego

SPPED WRITE 1

con este comando lo que habremos hecho es decirle al AMSTRAD que grabe a 2000 baudios. Asi la carga durará menos. Si no ejecutas ese comando, la grabación se realizará a 1000 baudios, más segura pero mucho mas lenta

save "LOADER.BAS"

me saldra que presione rec&play

y luego doy al enter

después grabar todos y cada uno de los ficheros:

```
SAVE “8BP.LIB”,b,27000,5000
SAVE “MUSIC.BIN”,b,32000,1500
SAVE “SPRITES.BIN”,b,33500,8500
SAVE “MONT7.BAS”
```

Por último, debemos hacer una última operación para que winape cierre el fichero.

file->remove tape

tras hacer el remove tape, el fichero adquirirá su tamaño (si no lo haces puedes ver que en el disco de tu PC el fichero no crece y es debido a que no se ha volcado al disco)

5 Ciclo de juego

Un videojuego de arcade, plataformas, aventuras, generalmente tiene un tipo de estructura similar, en la que unas ciertas operaciones se van a repetir cíclicamente en lo que denominaremos “ciclo de juego”.

En cada ciclo de juego actualizaremos posiciones de sprites e imprimiremos en pantalla los sprites, de modo que el número de ciclos de juego que se ejecutan por segundo equivale a los “fotogramas por segundo” (fps) del juego.

El siguiente pseudo código esquematiza la estructura básica de un juego

```
INICIO
Inicialización de variables globales: vidas, etc
Espera a que el usuario pulse tecla de comienzo de juego
GOSUB pantalla1
GOSUB pantalla 2
...
GOSUB pantalla N
GOTO INICIO

Código de PANTALLA N (siendo N cualquier pantalla)
        Inicialización de coordenadas de enemigos y personaje
        Pintado de la pantalla (layout), si procede

        BUCLE PRINCIPAL (ciclo del juego en esta pantalla)
                Lógica de personaje : Lectura de teclado y actualización de
                coordenadas del personaje y si procede, actualización de su
                secuencia de animación

                Ejecución de lógica de enemigo 1
                Ejecución de lógica de enemigo 2
                ...
                Ejecución de lógica de enemigo n

                Impresión de todos los sprites
                Condición de salida de esta pantalla (IF ... THEN RETURN)
                GOTO BUCLE PRINCIPAL
```

Fig. 7 Estructura básica de un juego

Si la lógica de los enemigos es muy pesada por haber muchos enemigos o por ser muy compleja, esto consumirá mas tiempo en cada ciclo del juego y por lo tanto el número de ciclos por segundo se verá reducido. Intenta no bajar de 10fps o 15 fps para que el juego mantenga un nivel de acción aceptable.

6 Sprites

6.1 Editar sprites con spedit y ensamblarlos

Spedit (Simple Sprite Editor) es una herramienta que te va a permitir crear tus imágenes de personajes y enemigos y usarlos en tus programas BASIC

Spedit está hecha en BASIC, y es muy sencilla, de modo que puedes modificarla para que haga cosas que no están contempladas y te interesen. Se ejecuta en el amstrad CPC, aunque esta pensada para que la utilices desde el emulador winape.

Lo primero que debes hacer es configurar winape para que la salida de la impresora la saque a un fichero. En este ejemplo he puesto la salida de la impresora al fichero printer5.txt

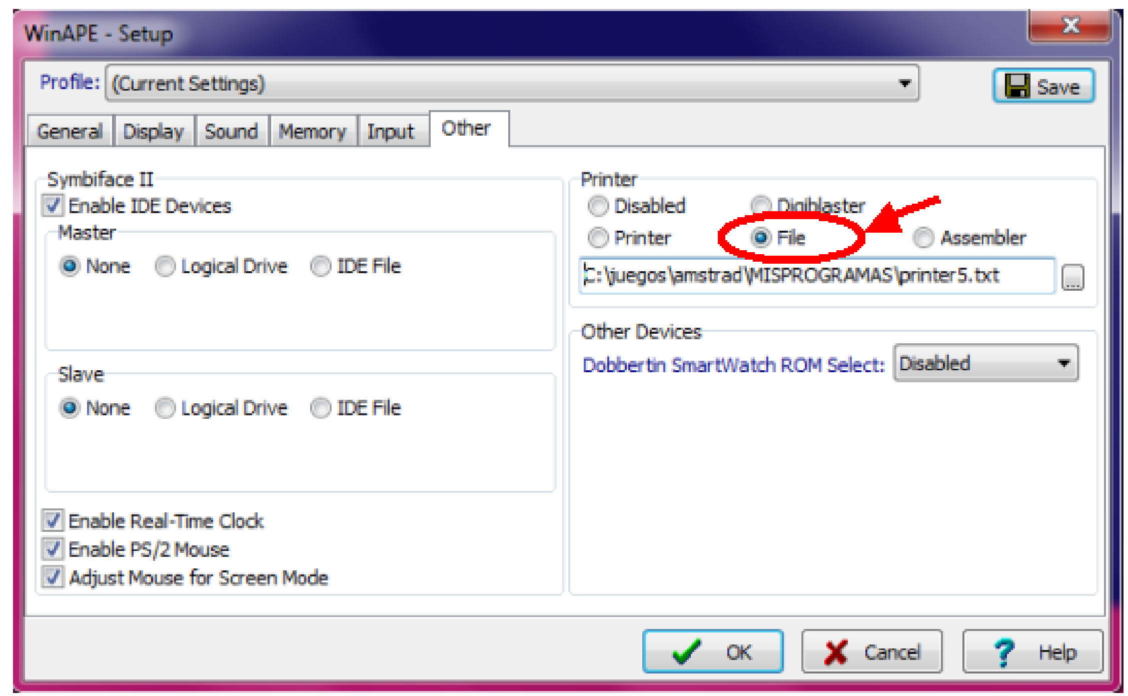


Fig. 8 Redireccion de la impresora del CPC a un fichero con Winape

Cuando ejecutes SPEDIT te aparecerá el siguiente menu, donde puedes elegir si vas a usar la paleta por defecto o bien una tuya que quieras definir. Si decides definir tu propia paleta, deberas reprogramar las líneas de BASIC donde se define la paleta alternativa, que es una subrutina a la que se invoca con GOSUB cuando pulsas “N” en la respuesta a la pregunta sobre si quieres usar la paleta por defecto.

```
SPEDIT: Simple Sprite Editor

 ---- CONTROLES EDICION----
  1,2 : tinta -/+
  qaop : mueve cursor pixel
  space: pinta
  h: flip horizontal
  v: flip vertical
  c: clear sprite
  b: imprime bytes (asm) en printer
  r: reload &4000
 -------------------------
Antes de empezar puedes cargar un sprite
 ensamblandolo en la direccion &4000
 No ensambles ancho y alto, tan solo los
 bytes del dibujo.

 La paleta la puedes cambiar pero debera
 ser la misma en tu programa
default paleta?(Y/N)y
ancho (par)?12
alto?24
```

Fig. 9 Pantalla ininical de SPEDIT

Suponiendo que eliges usar la paleta por defecto, la herramienta se pone en mode 0 y te permite editar dibujos, con la ayuda en la pantalla. Manejas un píxel que parpadea y en la parte inferior se muestra las coordenadas donde te encuentras y el valor del byte en el que te encuentras.



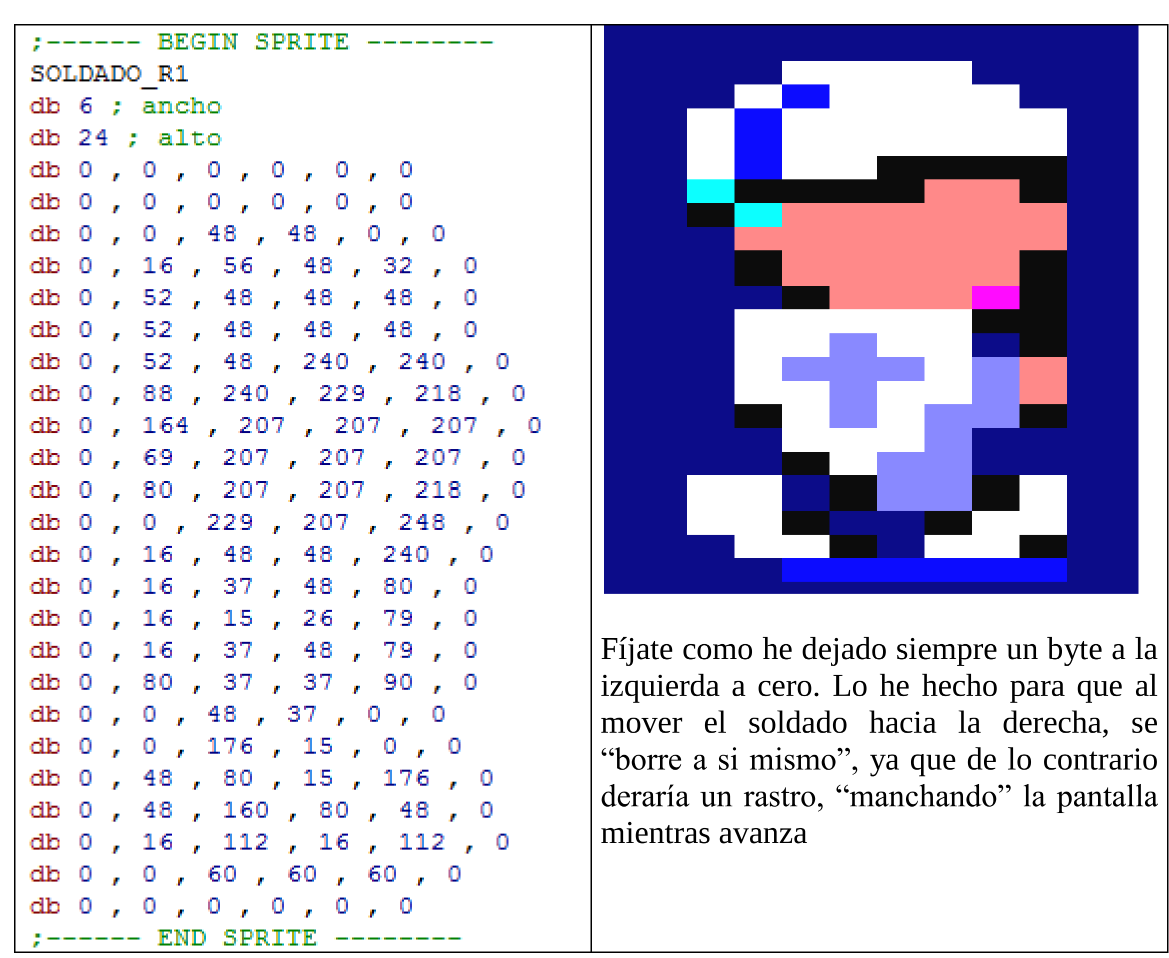
Fig. 10 Pantalla de edición de SPEDIT

SPEDIT te permite “espejar” tu imagen para hacer el mismo muñeco caminando hacia la izquierda sin esfuerzo, basta con pulsar H (flip horizontal) y lo mismo se puede hacer en vertical.

Es fácil adaptar esta herramienta para que te permita editar en mode 1 si lo deseas. De hecho es tan sencillo como eliminar la línea de BASIC que establece el modo de

pantalla. Como ves la herramienta no es del todo completa pero te permite hacer absolutamente todo lo que necesitas.

Una vez que has definido tu muñeco, para extraer el código ensamblador deberás pulsar la “b”. Esto mandará a la impresora (al fichero que hayamos definido como salida) un texto como el siguiente, al que puedes añadir un nombre, yo le he llamado “SOLDADO_R1”



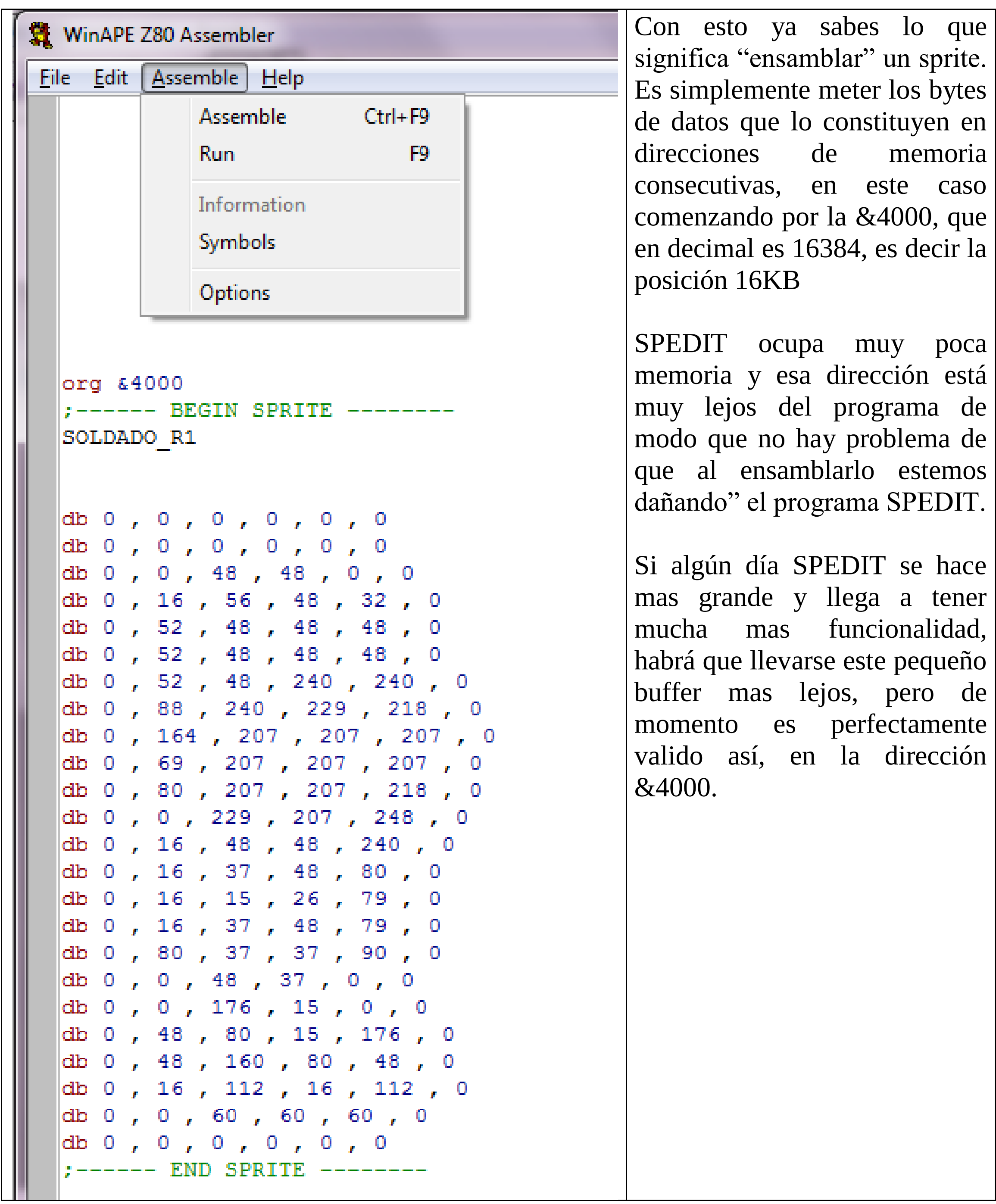
```
;------ BEGIN SPRITE --------
SOLDADO_R1
db 6 ; ancho
db 24 ; alto
db 0 , 0 , 0 , 0 , 0 , 0
db 0 , 0 , 0 , 0 , 0 , 0
db 0 , 0 , 48 , 48 , 0 , 0
db 0 , 16 , 56 , 48 , 32 , 0
db 0 , 52 , 48 , 48 , 48 , 0
db 0 , 52 , 48 , 48 , 48 , 0
db 0 , 52 , 48 , 240 , 240 , 0
db 0 , 88 , 240 , 229 , 218 , 0
db 0 , 164 , 207 , 207 , 207 , 0
db 0 , 69 , 207 , 207 , 207 , 0
db 0 , 80 , 207 , 207 , 218 , 0
db 0 , 0 , 229 , 207 , 248 , 0
db 0 , 16 , 48 , 48 , 240 , 0
db 0 , 16 , 37 , 48 , 80 , 0
db 0 , 16 , 15 , 26 , 79 , 0
db 0 , 16 , 37 , 48 , 79 , 0
db 0 , 80 , 37 , 37 , 90 , 0
db 0 , 0 , 48 , 37 , 0 , 0
db 0 , 0 , 176 , 15 , 0 , 0
db 0 , 48 , 80 , 15 , 176 , 0
db 0 , 48 , 160 , 80 , 48 , 0
db 0 , 16 , 112 , 16 , 112 , 0
db 0 , 0 , 60 , 60 , 60 , 0
db 0 , 0 , 0 , 0 , 0 , 0
;------ END SPRITE --------
```

Fíjate como he dejado siempre un byte a la izquierda a cero. Lo he hecho para que al mover el soldado hacia la derecha, se “borre a si mismo”, ya que de lo contrario deraría un rastro, “manchando” la pantalla mientras avanza

Fig. 11 Soldado en formato .asm

Una vez que has hecho el primer fotograma de tu soldado puedes dejar el trabajo y continuar otro día. Para partir del soldado que has dibujado y continuar retocándolo o bien modificarlo para construir otro fotograma, puedes ensamblar el soldado en la dirección &4000, quitando el ancho y el alto. Una vez ensamblado desde winape, le dices a SPEDIT que vas a editar un sprite del mismo tamaño y una vez estés en la pantalla de edición pulsas “r”. El sprite se cargará desde la dirección &4000, que es donde lo has “ensamblado”

Gran parte del atractivo de un juego son sus sprites. No escatimes tiempo en esto, hazlo despacio y con gusto y tu juego parecerá mucho mejor.



Con esto ya sabes lo que significa "ensamblar" un sprite. Es simplemente meter los bytes de datos que lo constituyen en direcciones de memoria consecutivas, en este caso comenzando por la &4000, que en decimal es 16384, es decir la posición 16KB

SPEDIT ocupa muy poca memoria y esa dirección está muy lejos del programa de modo que no hay problema de que al ensamblarlo estemos dañando" el programa SPEDIT.

Si algún día SPEDIT se hace mas grande y llega a tener mucha mas funcionalidad, habrá que llevarse este pequeño buffer mas lejos, pero de momento es perfectamente valido así, en la dirección &4000.

Fig. 12 Ensamblado de gráficos

Para saber en que dirección de memoria se ha ensamblado cada imagen, utiliza desde el menu de winape:

Assemble->symbols

Con ello veras una relacion de las etiquetas que has definido, como "SOLDADO_R1" y la dirección de memoria a partir de la cual se ha ensamblado.

Una vez que hayas hecho las diferentes fases de animación de tu soldado, puedes agruparlas en una "secuencia" de animación.

Las secuencias de animación son listas de imágenes y no se definen con SPEDIT. Con SPEDIT simplemente editas los “fotogramas”. En un apartado posterior te explicaré como decirle a la librería 8BP que un conjunto de imágenes constituyen una secuencia de animación.

Las imágenes que vayas haciendo para tu juego ve guardándolas todas en un único fichero, que se titule “images_mijuego.asm”, por ejemplo. Una vez que estén todas hechas podrás ensamblar ese fichero en la dirección 34000 y lo salvarás en un fichero binario desde el amstrad con el comando SAVE

Por ejemplo si has hecho 2000 bytes de imágenes, tras ensamblarlas en 34000 ejecuta desde el BASIC del CPC en el emulador:

SAVE “sprites.bin”, b, 34000,2000

Con esto habrás salvado en disco tu fichero de imágenes. No te olvides de la letra “b”, que sirve para especificar que se trata de un fichero binario.

Si quieres salvar las imágenes y las secuencias de animación juntas (las secuencias se encuentran en la 33500) simplemente ejecuta

SAVE “sprites.bin”, b, 33500,2500

Para cargar tus sprites en memoria RAM, simplemente ejecuta:

Load “sprites.bin”

6.2 Tabla de atributos de sprites

Los sprites se almacenan en una tabla que contiene un total de 32 sprites.
Cada entrada de la tabla contiene todos los atributos del sprite y ocupa 16 bytes por razones de rendimiento, ya que 16 es múltiplo de 2 y ello permite acceder a cualquier sprite con una multiplicación muy poco costosa. La tabla se encuentra ubicada en la dirección de memoria 27000, de modo que se puede acceder desde basic con PEEK y POKE, aunque tambien disponemos de comandos RSX para manipular los datos de esta tabla.

Los sprites tienen un conjunto de parámetros, de los que el primero de ellos es el byte de flags de status. En este byte, se usan 5 bits para representar un flag y cada flag significa una cosa, concretamente representan si el sprite se toma en consideración al ejecutar ciertas funciones
En la siguiente tabla se resume lo que ocurre si estan activos (a “1”)

Bit	Bit 4 (flag de movimiento relativo)	Bit 3 (flag de movimiento automático)	Bit 2 (flag de animación)	Bit 1 (flag de colision de sprites)	Bit 0 (flag de impresión)
efecto	MOVERALL() lo mueve	AUTOALL() lo mueve	ANIMALL() lo anima	COLSP() lo detecta	PRINTSPALL() lo imprime

Tabla 3 flags en el byte de estado

Para entender la potencia de estos flags vamos a ver algunos ejemplos:

Bit 0 flag de impresión: nuestro personaje o las naves enemigas lo tendrán activado y en cada ciclo del juego invocaremos a PRINTSPALL y se imprimirán todos a la vez

bit 1 flag de colision: una fruta o moneda por ejemplo pueden no tener flag de impresion pero tener el de colision

bit 2 flag de animacion automatica: se tiene en cuenta en ANIMA_ALL(). En el caso del personaje , recomiendo desactivarlo, ya que si me quedo quieto no hay que cambiar el fotograma.

bit 3 flag de movimiento automatico. Se mueve solo al invocar AUTO_ALL() teniendo en cuenta su velocidad. util en meteoritos y guardias que van y vienen.

bit 4 flag de movimiento relativo. todos los sprites que tengan este flag se mueven a la vez al invocar MOVER_ALL(incy,incx) muy util en naves en formacion y llegadas a planetas. También sirve para simular un scroll si dejas tu personaje en el centro y al pulsar los controles se desplazan casas o elementos de alrededor. Parecerá que es tu personaje el que avanza por un territorio.

Ejemplos de asignación del valor del byte de status:

Tipico enemigo: un sprite que se debe imprimir en cada ciclo, con deteccion de colision con otros sprites y animacion debe tener:

status = 1(bit 0)+ 2 (bit1) + 4 (bit 2) =7 =&x0111

Una casa que se desplaza al movernos: un sprite que se imprime en cada ciclo pero sin deteccion de colision con otros y movimiento relativo

status = 1(bit 0)+ 0 (bit1) + 0 (bit 2) + 0 (bit 3)+ 16 (bit 4)=17 =&x10001

Una fruta que nos da bonus: es un sprite que no se imprime en cada ciclo pero tiene deteccion de colisión

status = 0(bit 0)+ 2 (bit1) = 2 =&x10

La tabla de atributos de sprites se compone de 32 entradas de 16 bytes cada una, comenzando en la dirección 27000
El motivo de tener 16 bytes no es otro que el del rendimiento, ya que calcular la dirección del sprite N implica multiplicar por 16, lo cual al ser un múltiplo de 2, se puede hacer con un desplazamiento. Esto es útil en operaciones que involucran un único sprite. Para operaciones que recorren la tabla de sprites (como |PRINTSPALL o |COLSP) , internamente se recorre la tabla con un indice al que se le suma 16 para pasar de un sprite al siguiente. La suma es lo más rápido en ese caso.

Los atributos que tiene cada sprite son:

atributo	Byte	Logitud (bytes)	significado
status	0	1	Byte que contiene los flags de status para las operaciones PRINTSPALL, COLSP, ANIMALL, AUTOALL Y MOVERALL
Y	1	2	Coordenada Y [-32768..32768] los valores correspondientes al interior de la pantalla son [0..199]
X	3	2	Coordenada X en bytes [-32768..32768] los valores correspondientes al interior de la pantalla son [0..79]
Vy	5	1	Paso a dar en el movimiento automático
Vx	6	1	Paso a dar en el movimiento automático
Secuencia	7	1	Identificador de la secuencia de animación [0..31]. Si no posee secuencia se debe asignar un cero
Fotograma	8	1	Numero de frame en la secuencia [0..7]
Imagen	9	2	Dirección de memoria donde esta la imagen
Uso futuro	10	6	6 bytes para posibles funcionalidades futuras

La dirección de las coordenadas de cada sprite se pueden calcular asi

Dirección coordenada Y =27000 +16 *N +1
Dirección coordenada X =27000 +16 *N +3

De esta manera podremos hacer POKE un cambio en esa coordenada de cualquier sprite Y en general, las direcciones de los 32 sprites para manejar con POKE y PEEK son:

sprite	status	coordy	coordx	vy	vx	seq	frame	imagen
0	27000	27001	27003	27005	27006	27007	27008	27009
1	27016	27017	27019	27021	27022	27023	27024	27025
2	27032	27033	27035	27037	27038	27039	27040	27041
3	27048	27049	27051	27053	27054	27055	27056	27057
4	27064	27065	27067	27069	27070	27071	27072	27073
5	27080	27081	27083	27085	27086	27087	27088	27089
6	27096	27097	27099	27101	27102	27103	27104	27105
7	27112	27113	27115	27117	27118	27119	27120	27121
8	27128	27129	27131	27133	27134	27135	27136	27137
9	27144	27145	27147	27149	27150	27151	27152	27153
10	27160	27161	27163	27165	27166	27167	27168	27169
11	27176	27177	27179	27181	27182	27183	27184	27185
12	27192	27193	27195	27197	27198	27199	27200	27201
13	27208	27209	27211	27213	27214	27215	27216	27217
14	27224	27225	27227	27229	27230	27231	27232	27233
15	27240	27241	27243	27245	27246	27247	27248	27249
16	27256	27257	27259	27261	27262	27263	27264	27265
17	27272	27273	27275	27277	27278	27279	27280	27281
18	27288	27289	27291	27293	27294	27295	27296	27297
19	27304	27305	27307	27309	27310	27311	27312	27313
20	27320	27321	27323	27325	27326	27327	27328	27329
21	27336	27337	27339	27341	27342	27343	27344	27345
22	27352	27353	27355	27357	27358	27359	27360	27361
23	27368	27369	27371	27373	27374	27375	27376	27377
24	27384	27385	27387	27389	27390	27391	27392	27393
25	27400	27401	27403	27405	27406	27407	27408	27409
26	27416	27417	27419	27421	27422	27423	27424	27425
27	27432	27433	27435	27437	27438	27439	27440	27441
28	27448	27449	27451	27453	27454	27455	27456	27457
29	27464	27465	27467	27469	27470	27471	27472	27473
30	27480	27481	27483	27485	27486	27487	27488	27489
31	27496	27497	27499	27501	27502	27503	27504	27505

Tabla 4 Direcciones de atributos de los 32 sprites

Como ves las coordenadas X e Y son números de 2bytes. Los sprites aceptan coordenadas negativas por lo que puedes imprimir parcialmente un sprite en la pantalla, dando la sensación de que va entrando poco a poco. No podrás establecer coordenadas negativas con POKE , pero si podrás hacerlo con |LOCATESP y también con |POKE, que es una versión del comando POKE de BASIC pero que acepta números negativos.

Otro aspecto interesante son el espacio de memoria libre entre cada dos sprites. Por ejemplo el primer sprite acaba en la dirección 27010 (la imagen ocupa 2 bytes) y el segundo empieza en la dirección 27016, por lo que hay 5 bytes que sobran y que se podrían utilizar en el futuro para nuevas funcionalidades de la librería 8BP.

6.3 Colisiones entre sprites

Para comprobar si tu personaje o tu disparo ha colisionado con otros sprites dispones del comando

|COLSP,<sprite_number>,@colision%

Donde sprite number es el sprite que quieres comprobar (tu personaje o tu disparo) y la variable colision es una variable entera que previamente ha tenido que ser definida, asignando un valor inicial, por ejemplo:

colision%=0
|COLSP,1,@colision%

La variable colisión se rellenará con el primer identificador de sprite que se detecte que ha colisionado con tu sprite, aunque podría ocurrir una colisión múltiple, pero el comando solo te entrega un resultado.

Internamente la librería 8BP recorre los sprites desde el 31 hasta el 0, y si tienen el flag de colisión activo (bit 1 del byte de status) entonces se comprueba si colisiona con tu sprite. Si no hay colisión, la variable colision% queda con valor cero. En caso de haberla retornará el número de sprite que esté colisionando con tu sprite. Si por ejemplo colisionan el 4 y el 12, la función retornará un 12 pues comprueba antes el 12 que el 4.

6.4 Tabla de secuencias de animación

Las animaciones suelen componerse de un numero par de fotogramas, aunque esto no es una regla estricta. Piensa por ejemplo en la animación simple de un personaje con solo dos fotogramas: piernas abiertas y cerradas. Son dos fotogramas. Ahora piensa en una animación mejorada, con una fase de movimiento intermedia. Esto supone crear la secuencia: cerradas-intermedia-abiertas-intermedia- y vuelta a empezar. Como ves es numero par, son 4

Las secuencias de animación de 8BP son listas de 8 fotogramas, no pueden tener mas, aunque siempre puedes hacer secuencias mas cortas.

Los fotogramas de una secuencia de animación son las direcciones de memoria donde están ensambladas las imágenes de las que se componen, pudiendo ser diferentes en tamaño, aunque lo normal es que sean iguales. Si a mitad de la secuencia introduces un cero, el significado es que la secuencia ha terminado. Veamos un ejemplo en lenguaje ensamblador aunque también la puedes crear usando el comando |SETUPSQ

```
dw MONTOYA_R0,MONTOYA_R1,MONTOYA_R2,MONTOYA_R1,0,0,0,0

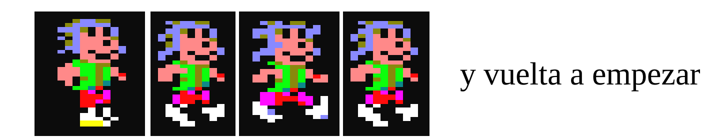
el equivalente en BASIC usando la librería 8BP es

SETUPSQ,1, &926c,&92FE,&9390 ,&02fe ,0,0,0,0
```

Nótese que en BASIC requieres conocer las diferentes direcciones de memoria en las que se ha ensamblado cada imagen. Ello lo puedes ver desde el menu de winape Assemble->symbols

En ensamblador usas directamente las etiquetas de cada imagen, por lo que es más“entendible”. Pocas veces el ensamblador es mas fáil de entender que el BASIC!!!

se trata de una secuencia de animación de 3 fotogramas diferentes pero para que sea fluida antes de volver a empezar hay que pasar por el fotograma “intermedio” otra vez, de modo que al final son 4 fotogramas:



y vuelta a empezar

Fig. 13 secuencia de animación

Si quisieses hacer una secuencia de mas de 8 fotogramas podrías simplemente encadenar dos secuencias seguidas y cuando el personaje llegase al ultimo fotograma de la primera secuencia usar el comando |SETUPSP para asignarle la segunda secuencia

Las secuencias de animación se ensamblan a partir de la dirección 33500, de modo que se dispone de 500 bytes antes de llegar a la dirección 34000 que es donde se ensamblan los gráficos. Por lo tanto dispones de espacio para almacenar hasta 31 secuencias de animación.

Cada secuencia almacena 8 direcciones de memoria correspondientes a los 8 fotogramas, esto son 16 bytes. En 500 bytes hay espacio para 31 secuencias y sobran 4 bytes.

31 * 16 = 496

tu fichero de secuencias de animación se puede parecer a esto:

```
org 33500;
;======================================================================
; 31 secuencias de animacion (500bytes) de 8 frames
;======================================================================
; debe ser una tabla fija y no variable
; cada secuencia contiene las direcciones de frames de animacion
ciclica
; cada secuencia son 8 direcciones de memoria de imagen
; numero par porque las animaciones suelen ser un numero par
; un cero significa fin de secuencia, aunque siempre se
; gastan 8 words /secuencia
; al encontrar un cero se comienza de nuevo.
; si no hay cero, tras el frame 8 se comienza de nuevo
; en total caben 31 secuencias diferentes (disponemos de 500 bytes)
; SEQUENCES:
; la secuencia cero es que no hay secuencia.
; empezamos desde la secuencia 1

;--------------secuencias de animacion del personaje --------
dw MONTOYA_R0,MONTOYA_R1,MONTOYA_R2,MONTOYA_R1,0,0,0,0       ;1
dw MONTOYA_UR0,MONTOYA_UR1,MONTOYA_UR2,MONTOYA_UR1,0,0,0,0  ;2
```

```
dw MONTOYA_U0,MONTOYA_U1,MONTOYA_U0,MONTOYA_U2,0,0,0,0      ;3
dw MONTOYA_UL0,MONTOYA_UL1,MONTOYA_UL2,MONTOYA_UL1,0,0,0,0  ;4
dw MONTOYA_L0,MONTOYA_L1,MONTOYA_L2,MONTOYA_L1,0,0,0,0      ;5
dw MONTOYA_DL0,MONTOYA_DL1,MONTOYA_DL2,MONTOYA_DL1,0,0,0,0  ;6
dw MONTOYA_D0,MONTOYA_D1,MONTOYA_D0,MONTOYA_D2,0,0,0,0      ;7
dw MONTOYA_DR0,MONTOYA_DR1,MONTOYA_DR2,MONTOYA_DR1,0,0,0,0  ;8

;--------------secuencias de animacion del soldado --------
dw SOLDADO_R0,SOLDADO_R2,SOLDADO_R1,SOLDADO_R2,0,0,0,0 ;9
dw SOLDADO_L0,SOLDADO_L2,SOLDADO_L1,SOLDADO_L2,0,0,0,0 ;10
```

La librería 8BP te proporciona un comando llamado |SETUPSQ con el que puedes crear secuencias de animación desde BASIC. Dicho comando lo que hace realmente es meter datos en las direcciones de memoria destinadas a las secuencias (desde la 33500 hasta la 34000). Si las creas y las ensamblas y las salvas en el fichero de imágenes te ahorrarás tener que crearlas desde BASIC y por lo tanto ahorrarás líneas de BASIC.

7 Tu primer juego sencillo

Ya tienes los conocimientos para intentar un primer paso en la creación de videojuegos. Para ello vamos a ver un sencillo ejemplo de un soldado al que vas a controlar, haciendole caminar a derecha e izquierda por la pantalla

Supongamos que hemos editado a un soldado, gracias a SPEDIT. Y hemos construido sus secuencias de animación, las cuales han quedado con el identificador 9 y 10 para las direcciones de movimiento derecha e izquierda respectivamente.
Las dos secuencias de animación las hemos creado bien desde el fichero de secuencias.asm o bien en basic con el comando |SETUPSQ (el cual no he incluido en este listado)

```
10 MEMORY 26999
20 MODE 0: DEFINT A-Z: CALL &6B78:' install RSX
25 call &bc02:'restaura paleta por defecto por si acaso
26 ink 0,0:'fondo negro
30 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT:'reset sprites
40 |SETLIMITS,12,80,0,186: ' establecemos los limites de la pantalla
de juego
50 x=40:y=100:' coordenadas del personaje
51|SETUPSP,0,0,&1:' status del personaje
52|SETUPSP,0,7,9:'secuencia de animacion asignada al empezar
53|LOCATESP,0,y,x:'colocamos al sprite (sin imprimirlo aun)

60 'ciclo de juego
70 gosub 100
80 |PRINTSPALL,0,0
90 goto 60

99 ' rutina movimiento personaje -------------
100 IF INKEY(27)=0 THEN IF dir<>0 THEN |SETUPSP,0,7,9:dir=0:return
ELSE |ANIMA,0:x=x+1:GOTO 120
110 IF INKEY(34)=0 THEN IF dir<>1 THEN |SETUPSP,0,7,10:dir=1:return
ELSE |ANIMA,0:x=x-1
120 |LOCATESP,0,y,x
130 RETURN
```

Con este listado ya tienes un minijuego que te permite controlar un soldado y hacerlo corretear horizontalmente. Fíjate que si al caminar hacia la izquierda sobrepasas el valor minimo del limite establecido con SETLIMITS, se producirá el “clipping” del personaje, mostrándose tan solo la parte que queda dentro del area de juego permitida

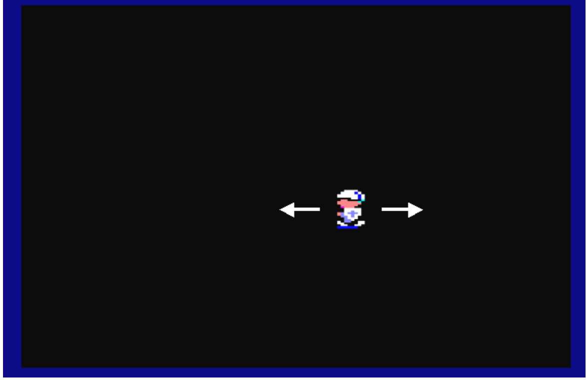


Tabla 5 Un sencillo juego

8 Juegos de pantallas: layout

8.1 Definición y Uso del layout

A menudo querrás que tus juegos consistan en un conjunto de pantallas donde el personaje deba recoger tesoros o esquivar enemigos en un laberinto. En esos casos se hace indispensable el uso de una matriz donde definas los bloques constituyentes de cada "laberinto" o tambien llamado "layout" de la pantalla

En la librería 8BP tienes un mecanismo muy sencillo para hacerlo, que además de proporciona una función de colisión para que compruebes si tu personaje se ha desplazado a una zona ocupada por un "ladrillo".

En 8BP un layout se define con una matriz de 20x25 "bloques" de 8x8 pixeles, los cuales pueden estar ocupados o no. Es decir, hay tantos bloques como tiene la pantalla de caracteres en mode 0.

Para imprimir un layout en la pantalla dispones del comando:

|LAYOUT,<y>,<x>,@string

Esta rutina imprime una fila de sprites para construir el layout o "laberinto" de cada pantalla. La matriz o "mapa del layout" se almacena en una zona de la memoria que maneja 8BP de modo que cuando imprimes bloques en realidad no solo estás imprimiendo en la pantalla, sino que tambien estas rellenando el area de memoria que ocupa el layout (20x25 bytes) donde cada byte representa un bloque.

Las coordenadas y,x se pasan en formato caracteres, es decir

y toma valores [0,24]
x toma valores [0,19]

Los bloques que imprime la funcion |LAYOUT se construyen con cadenas de caracteres y cada carácter se corresponde con un sprite que debe existir. De este modo el bloque "Z" se corresponde con la imagen que tenga asignada el sprite 31. El bloque "Y" se corresponde con la imagen que tenga asignada el sprite 30, y asi sucesivamente (consulta la tabla de conversión de caracteres a sprites en el capítulo de guia de referencia).
El @string es una variable de tipo cadena. no puedes pasar directamente la cadena. Es decir, sería ilegal algo como:

|LAYOUT,1,0,"ZZZ YYY"

lo correcto es:
cadena="ZZZ YYY"
|LAYOUT,1,0,@cadena

Ten cuidado de que la cadena no esté vacia, de lo contrario puede bloquearse el ordenador!!

Además, debes anteponer el símbolo “@” en la variable de tipo string para que la librería pueda ir a la dirección de memoria donde se almacena la cadena y asi poder recorrerla, imprimiendo uno a uno los sprites correspondientes.
Debes tener en cuenta que los espacios en blanco significan ausencia de sprite, es decir, en las posiciones correspondientes a los espacios no se imprime nada. Si había previamente algo en esa posición, no se borrará. Si deseas borrar necesitas definirte un sprite de borrado de 8x8, donde todo sean ceros.

Aunque usas los sprites para imprimir el layout, justo después de imprimirlo puedes redefinir los sprites con |SETUPSP y asignarles imágenes de soldados, monstruos o lo que quieras, es decir, el layout se “apoya” en el mecanismo de sprites para imprimir pero no te limita el número de sprites, pues dispones de los 32 para que sean lo que tu quieras justo después de imprimir el layout

Para detectar colisiones con el layout dispones de la función

|COLAY,<sprite number>, @colision%

Dado un sprite y dependiendo de sus coordenadas y de su tamaño, esta función averiguará si está colisionando con el layout y te avisará a través de la variable colision%, la cual debe estar previamente definida. Y no sólo debe estar previamente definida, sino que debes poner el “%” para indicar que es una variable entera, aunque estes usando DEFINT A-Z

Ejemplo de uso:

```
col%=0
|COLAY,0,@col%
```

Si no hay colision, la variable tomará el valor cero. Si hay colisión, tomará el valor 1

Vamos a ver un ejemplo de creación de un layout y de movimiento de un personaje dentro del layout, corrigiendo su posición si ha colisionado.

8.2 Ejemplo de juego con layout

Vamos a evolucionar un poco el juego presentado en el anterior capítulo, en lo que respecta al control del personaje. Esta vez vamos a usar a Montoya como ejemplo, el cual tiene 8 secuencias de animación, cada una para moverse en una dirección diferente. A las secuencias de animación les hemos asignado un número que va del 1 al 8.

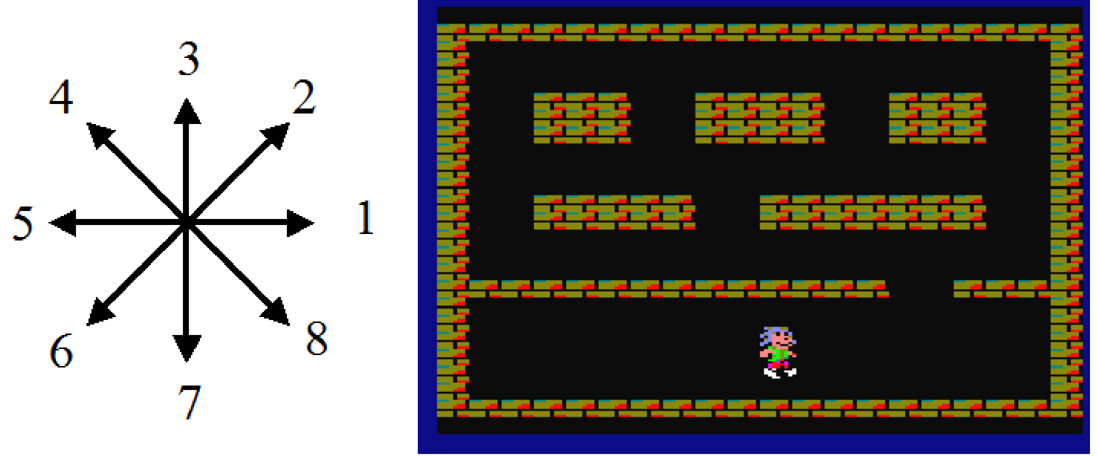


Tabla 6 Uso del layout en un juego

En la rutina de control del personaje hemos incluido colisión con el layout. En funcion de la dirección en la que avanzamos, modificamos las coordenadas “nuevas” (yn , xn) e invocamos a la funcion de colision con layout |COLAY,0 para chequear si el sprite 0 (nuestro personaje) ha colisionado. Si ha colisionado, corregimos las coordenadas (una o las dos) para dejarle en una posición sin colisión antes de imprimirle de nuevo

```
10 MEMORY 26999
20 MODE 0: DEFINT A-Z: CALL &6B78:' install RSX
25 call &bc02:'restaura paleta por defecto por si acaso
26 ink 0,0:'fondo negro
30 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT:'reset sprites
40 |SETLIMITS,0,80,0,200: ' establecemos los limites de la pantalla de
juego
50 dim c$(25):for i=0 to 24:c$(i)=" ":next
100 c$(1)= "ZZZZZZZZZZZZZZZZZZZZ"
110 c$(2)= "Z                  Z"
120 c$(3)= "Z                  Z"
125 c$(4)= "Z                  Z"
130 c$(5)= "Z  ZZZ  ZZZZ  ZZZ  Z"
140 c$(6)= "Z  ZZZ  ZZZZ  ZZZ  Z"
150 c$(7)= "Z  ZZZ  ZZZZ  ZZZ  Z"
160 c$(8)= "Z                  Z"
170 c$(9)= "Z                  Z"
190 c$(10)="Z                  Z"
195 c$(11)="Z  ZZZZZ  ZZZZZZZ  Z"
200 c$(12)="Z  ZZZZZ  ZZZZZZZ  Z"
210 c$(13)="Z                  Z"
220 c$(14)="Z                  Z"
230 c$(15)="Z                  Z"
240 c$(16)="ZZZZZZZZZZZZZZ  ZZZZ"
250 c$(17)="Z                  Z"
260 c$(18)="Z                  Z"
270 c$(19)="Z                  Z"
271 c$(20)="Z                  Z"
272 c$(21)="Z                  Z"
273 c$(22)="Z                  Z"
274 c$(23)="ZZZZZZZZZZZZZZZZZZZZ"
```

```
300 gosub 550: ' imprime el layout
310 xa=40:xn=xa:ya=150:yn=ya:' coordenadas del personaje
311|SETUPSP,0,0,&x111:' deteccion de colision con sprites y layout
312|SETUPSP,0,7,1: ' secuencia = 1
320 |LOCATESP,0,ya,xa: 'colocamos al personaje (sin imprimirlo)
325 cl%=0: 'declaramos la variable de colision, explicitamente entera
(%)


330 '----- ciclo de juego ----------------------
340 gosub 1500:'rutina de lectura teclado y movimiento de personaje
350|PRINTSPALL,0,0
360 goto 340

550 'rutina print layout-----------------------------------------
560 FOR i=0 TO 23:|LAYOUT,i,0,@c$(i):NEXT
570 RETURN

1500 ' rutina movimiento personaje -------------
1510 IF INKEY(27)<0 GOTO 1520
1511 IF INKEY(67)=0 THEN IF dir<>2 THEN |SETUPSP,0,7,2:dir=2:GOTO 1533
ELSE |ANIMA,0:xn=xa+1:yn=ya-2:GOTO 1533
1512 IF INKEY(69)=0 THEN IF dir<>8 THEN |SETUPSP,0,7,8:dir=8:GOTO 1533
ELSE |ANIMA,0:xn=xa+1:yn=ya+2:GOTO 1533
1513 IF dir<>1 THEN |SETUPSP,0,7,1:dir=1:GOTO 1533 ELSE
|ANIMA,0:xn=xa+1:GOTO 1533
1520 IF INKEY(34)<0 GOTO 1530
1521 IF INKEY(67)=0 THEN IF dir<>4 THEN |SETUPSP,0,7,4:dir=4:GOTO 1533
ELSE |ANIMA,0:xn=xa-1:yn=ya-2:GOTO 1533
1522 IF INKEY(69)=0 THEN IF dir<>6 THEN |SETUPSP,0,7,6:dir=6:GOTO 1533
ELSE |ANIMA,0:xn=xa-1:yn=ya+2:GOTO 1533
1523 IF dir<>5 THEN |SETUPSP,0,7,5:dir=5:GOTO 1533 ELSE
|ANIMA,0:xn=xa-1:GOTO 1533
1530 IF INKEY(67)=0 THEN IF dir<>3 THEN |SETUPSP,0,7,3:dir=3:GOTO 1533
ELSE |ANIMA,0:yn=ya-4:GOTO 1533
1531 IF INKEY(69)=0 THEN IF dir<>7 THEN |SETUPSP,0,7,7:dir=7:GOTO 1533
ELSE |ANIMA,0:yn=ya+4:GOTO 1533
1532 RETURN
1533 |LOCATESP,0,yn,xn:ynn=yn:|COLAY,0,@cl%:IF cl%=0 THEN 1536
1534 yn=ya:|POKE, 27001,yn:|COLAY,0,@cl%:IF cl%=0 THEN 1536
1535 xn=xa: yn=ynn:|POKE, 27001,yn:|POKE, 27003,xn:|COLAY,0,@cl%:IF
cl%=1 THEN yn=ya:|POKE,27001,yn
1536 ya=yn:xa=xn
1537 RETURN
```

8.3 Cómo abrir una compuerta en el layout

Si deseas que tu personaje pueda coger una llave y abrir una compuerta o en general eliminar una parte del layout para permitir el acceso, lo que tienes que hacer son dos pasos

1) Tener definido un sprite de borrado de 8x8 donde todo sean ceros. Usando |LAYOUT lo imprimes en las posiciones que deseas

2) A continuación, usando nuevamente |LAYOUT, imprimes espacios donde has borrado. Asi el map layout quedará con el carácter “ “ en esas posiciones y la función de colisión con el layout resultará cero

En el juego “mutante montoya” se utiliza esta técnica para abrir la puerta del castillo, asi como para abrir las compuertas que conducen a la princesa

En el siguiente ejemplo se ilustra el concepto, abriendo una compuerta situada en las coordenadas (10, 12) de un tamaño de 2 bloques, al coger una llave que esta definida con el sprite 16.
Nada mas coger la llave se abre la compuerta y la llave queda desactivada para no evaluar más veces la colisión con ella, es decir, el comando COLSP retornará un 32 a partir del momento que cojas la llave si vuelves a colisionar con ella.
Tras abrir la compuerta, si desplazas el personaje hasta el lugar que ocupaba dicha compuerta, la colision con layout dará como resultado 0

```
----- esta parte esta dentro del bucle de logica ----
6410 |PRINTSPALL,1,0
6411 |COLSP,0,@cs%:IF cs%<32 THEN IF cs%>=15 then gosub 6500
(... mas instrucciones . . .)
---------------------------------

-------- rutina de apertura de compuerta----------------
6499' se comprueba que tu colision sea con la llave, que es el sprite
16
6500 borra$="MM":spaces$="  ":' el sprite de borrado se ha definido
como "M" (M es el sprite 18 en el “idioma” del comando |LAYOUT)
6501 if cs%=16
then|LAYOUT,10,12,@borra$:|LAYOUT,10,12,@spaces$:|SETUPSP,16,0,0
6502 return
```

8.4 Cómo ahorrar memoria en tus layouts

Si tu juego tiene muchas pantallas y necesitas ahorrar espacio puedes utilizar una técnica sencilla para ahorrar memoria
Imagínate que solo hay un tipo de “ladrillo” en una pantalla (un ladrillo o ausencia del mismo). Esto se puede representar con un solo bit, de modo que en un byte caben 8 ladrillos. Puesto que no podemos escribir todos los códigos ASCII porque muchos de ellos son de control, al menos puedes usar la mitad. Ello “compactaría” la pantalla en una proporción de 1:7 (en el espacio de 10 pantallas podrás meter 70 pantallas) y simplemente antes de invocar a |LAYOUT deberás hacer la conversión entre tus bits (que estarán en forma de caracteres) y los caracteres que espera recibir el comando. Esa conversión la harías en BASIC y aunque sea lenta hablamos de imprimir la pantalla, lo cual no requiere excesiva velocidad.
La misma filosofia se puede aplicar si solo hay 4 tipos de ladrillos (dos tipos y ausencia). Puedes usar solo 2 bits por ladrillo y luego meter esos bits en caracteres (de 1 a 128), por lo que te caben 3.5 ladrillos por carácter.

en ese caso la proporción de ahorro es 2:7 es decir , de 3.5 veces menos. En el espacio de 10 pantallas podrás meter 35 pantallas.

Otra estrategia más avanzada y original sería diseñar un programa generador de layouts a partir de un número, de modo que mediante un algoritmo pudieses construir un layout completo sin necesidad de almacenar el layout, sino tan solo ese número que sirviese como “semilla” para construir layouts. Obviamente esta estrategia requiere de cierto ingenio.

9 Recomendaciones y programación avanzada

9.1 Recomendaciones de velocidad

El interprete BASIC es muy pesado en ejecución debido a que no solo ejecuta cada comando sino que analiza el número de línea, realiza un análisis sintáctico del comando introducido, valida su existencia, el número y tipo de parámetros, que sus valores q¡se encuentren en rangos validos (por ejemplo PEN 40 es ilegal) y muchas mas cosas. Es el análisis sintáctico y semántica de cada comando lo que realmente pesa y no tanto su ejecución. El caso de los comandos RSX no es una excepción. El interprete BASIC comprueba su sintaxis y eso pesa mucho, a pesar de que sean rutinas escritas en ASM, pues antes de invocarlas, el intérprete BASIC ya ha hecho muchas cosas

Por consiguiente hay que ahorrar ejecuciones de comandos, programando con astucia para que la lógica del programa pase por el menor numero de instrucciones posibles, aunque ello a veces implique escribir más instrucciones. El uso de GOTO es muy recomendable, dada su elevada velocidad de ejecución, como veremos mas adelante en una tabla comparativa

Un factor decisivo a la hora de invocar un comando es el paso de parámetros. Cuantos mas parámetros tiene, mas costoso es su interpretación por parte del BASIC, incluso aunque sea una rutina ASM que se invoque por CALL, pues el comando CALL sigue siendo BASIC y antes de acceder a la rutina en ASM, se analiza el nÚmero y tipo de parámetros irremediablemente.

Para evaluar el coste de ejecución de un comando puedes usar el siguiente programa. También te servirá para evaluar el rendimiento de nuevas funciones en ensamblador que incorpores a la librería 8BP si deseas hacerlo.

```
10 MEMORY 26999
20 DEFINT a-z
30 a!= TIME
40 FOR i=1 TO 1000
50  <aqui pones un comando, por ejemplo PRINT “A”>
60 NEXT
70 b!=TIME
80 PRINT (b!-a!)
900 c!=1000/((b!-a!)*1/300)
100 PRINT c, "fps"
110 d!=c!/60
120 PRINT "puedes ejecutar ",d!, "comandos por barrido (1/50 seg)"
125 rem si dejas la linea 50 vacia , tardara 0.47 milisegundos
130 PRINT "el comando tarda ";((b!-a!)/300-0.47);"milisegundos"
```

Vamos a ver a continuación el resultado del rendimiento de algunos comandos. Hay que decir que es más rápido ejecutar una llamada directa a la dirección de memoria (un CALL &XXXX) que invocar el comando RSX. En la siguiente tabla obviamente cuanto menor sea el resultado (expresado en milisegundos), mas rápido es el comando. La

tabla que aquí se presenta debes tenerla en todo momento presente y tomar tus decisiones de programación en base a ella.

Comando	ms	Comentario
PRINT “A”	3.63	Lentísimo. Ni se te ocurra usarlo, salvo puntualmente para cambiar el numero de vidas pero no imprimas puntuación en un juego por cada enemigo que mates
LOCATE 1,1 PRINT puntos	24.87	Colocar el cursor de texto con LOCATE e imprimir el valor de luna variable “puntos” es costosísimo. Si actualizas puntos hazlo sólo de vez en cuando y no en cada ciclo de juego
REM hola	0.20	Los comentarios consumen
‘ hola	0.25	Ahorras 2 bytes de memoria pero es mas lento!!
GOTO 60	0.196	Muy rápido!!! Mas rápido incluso que REM. Usa este comando sin piedad, úsalo!!!
|NOP	1.17	No hace nada, solo es un RET de ensamblador. Nos da una idea de lo que tarda el analizador sintáctico del BASIC. Este es el tiempo mínimo que va a tardar en ejecutarse cualquier comando RSX
|NOP,1,2,3	2.15	El paso de parámetros es algo costoso
|LOCATESP,i,y,x	2.47	Es muy rápida teniendo en cuenta la máxima velocidad alcanzable (2.15 es lo que tarda NOP con 3 parámetros) pero si no usas coordenadas negativas es mejor usar el comando BASIC POKE. S quieres ubicar un escuadrón de enemigos es mejor usar AUTOALL y MOVERALL, ya que si por cada enemigo usas un LOCATESP invertirás demasiado tiempo.
POKE &XXXX, valor	0.71	Muy rápido! Úsalo para actualizar las coordenadas de los sprites
POKE d,valor	0.85	Muy rápido teniendo en cuenta que debe traducir la variable “d”
|POKE,&xxxx,valor	1.87	Permite números negativos y si solo actualizas una coordenada (X o Y) es mejor que LOCATESP
X=PEEK(&xxxx)	0.93	Muy rápido!
X=INKEY(27)	1.12	Muy rápido. Apto para videojuegos aunque debes usarlo inteligentemente como se recomienda en este libro.
IF x>50 THEN x=0	1.42	Cada IF pesa, hay que tratar de ahorrarlos porque una lógica de juego va a tener muchos
If inkey(27)=1 then x=1	1.75	Buen uso combinado. Es más rápido que hacer b=INKEY(27) y después el IF..THEN
:	0.05	No ahorra mucho pero es mas rápido usar “:” en lugar de un nuevo número de línea, y si aplicas esto muchas veces acabas ahorrando de forma significativa
|PRINTSP,0,10,10	5.4	Un solo sprite de 14 x 24 . Ojo, si vas a imprimir varios compensa mucho mas imprimir todos los sprites de golpe con PRINTSPALL
CALL &xxxx,0,10,10		Equivalente a PRINTSP, así es mas rápido aunque menos legible
|PRINTSPALL,0,0	20.6	Imprimiendo 6 sprites grandes, casi puede con todos en

(6 sprites 14x24)		cada barrido de pantalla ya que 20.6 x 50 =1030 ms y un segundo son 1000 ms Usando "CALL" tarda exactamente 20.0 ms y por lo tanto puede con los 6
|PRINTSPALL,0,0 (32 sprites 12x16)	70.4	Esto significan unos 14fps a plena carga de sprites. Lo que tarda es T = 3.14 + n x 2.1 Es decir, unos 2ms por sprite y un coste fijo de 3.14ms. Este coste fijo coincide con el número PI pero es pura casualidad. Es el coste del análisis sintáctico de BASIC sumando al de recorrer la tabla de sprites buscando cuales hay que imprimir.
|COLAY,0,@x%	3.44	Aceptable. Usar solo con el personaje, no con los enemigos o el juego ira lento. Si el personaje mide múltiplos de 8 es más rápido. En este ejemplo era de 14x24 y lógicamente 14 no es múltiplo de 8. cuanto mayor es el sprite mas tarda
CALL &XXXX,0,@x%	2.79	Es como invocar a |COLAY . mas rápido pero menos legible
GOSUB / RETURN	0.56	Aceptablemente rápido. La prueba es con una rutina que solo hace return.
|SETUPSP, id, param, valor	3.5	Aceptable, aunque POKE es mucho mejor. Para ciertas cosas SETUPSP es mas legible y normalmente se va a usar poco en la lógica de modo que no es un problema usarlo (por ejemplo al cambiar de dirección un sprite). En el comando SETUPSP vemos el numero de sprite y parámetro que se toca. En POKE no se ve nada, es menos legible aunque mas rápido.
FOR / NEXT	0.6	Lo puedes usar para recorrer varios enemigos y que cada uno se mueva de acuerdo a una misma regla. Debes valorar si puedes usar AUTOALL o MOVEALL para tus propósitos ya que en un solo comando moverías a todos los que quieras, lo cual es mucho mejor que un bucle.
|COLSP,0,@c%	4.97	Tarda lo mismo con independencia del numero de sprites activos. Esta rutina la tendrás que invocar en cada ciclo de la lógica de tu juego, de modo que son casi 5ms que obligatoriamente tienes que destinar a esto. Una estrategia como ejecutar |COLSP tan sólo la mitad de los frames no funcionaría porque requiere incrementar un contador y comprobarlo con un IF, por lo que aunque ahorras 2.5ms los gastarías en comprobaciones.
|ANIMALL	3.0	Es algo costoso pero hay una forma de invocarlo conjuntamente al invocar |PRINTSPALL , mediante un parámetro que hace que se invoque a esta función antes de imprimir los sprites. Ello permite ahorrar la capa del BASIC , es decir lo que consume enviar el comando, que es de 1.17ms (comando |NOP). Por ello podemos decir que este comando consumirá normalmente algo menos de

		2ms
|AUTOALL	4.25	Es costosa pero puede mover a la vez los 32 sprites
|MOVERALL,1,1	5.14	Es costosa pero puede mover a la vez los 32 sprites
SOUND	10	El comando sound es “bloqueante” en cuanto se llena el buffer de 5 notas. Esto significa que tu lógica de BASIC no debe encadenar mas de 5 comandos SOUND o se parará hasta que alguna nota termine. En cualquier caso si decides usarlo debe ser con sumo cuidado ya que consume mucho tiempo su ejecución (10 ms es muchísimo)

Tabla 7 Relacion de tiempos de ejecución de instrucciones

Cuando hagas tu programa, trata de ver el coste de los comandos que usas y minimiza al máximo el uso de la CPU. Por ejemplo si puedes evitar pasar por un IF insertando un GOTO, siempre será preferible. O haz uso de POKE en lugar de LOCATESP a menos que uses coordenadas negativas. Cuando te falte velocidad y necesites un poquito mas de rapidez utiliza CALL y abandona el uso de RSX

Ten en cuenta que toda la lógica de tu programa debe acumular como mucho 20ms si quieres sincronizar con los barridos de pantalla, aunque la jugabilidad se mantendrá aceptable muy por encima, quizás hasta los 50ms, depende del tipo de juego. Es muy difícil que logres 20ms a menos que no haya apenas sprites. Pero un objetivo de 50ms es razonablemente rápido. Si tu juego tarda 50ms entonces generará 20fps (frames por segundo) y será muy aceptable. Y si consigues 40ms (25fps) incluso lograrás una suavidad profesional en los movimientos.

Más recomendaciones importantes:

Usar DEFINT A-Z al principio del programa. El rendimiento mejorará muchísimo. Esto es casi obligatorio. Este comando borra las variables que existiesen antes y obliga a que todas las nuevas variables sean enteros a menos que se indique lo contrario con modificadores como “$” o “!” (consulta la guia de referencia de programador BASIC de amstrad)
Eliminar espacios en blanco
Eliminar cualquier comentario en la lógica de juego y si dejas alguno que sea REM (mas rápido), no uses la comilla. Si usas la comilla es para ahorrar 2 bytes de memoria, y es adecuado para comentar el resto del programa (inicializaciones y cosas asi). Si quieres comentar partes de lógica puedes hacer lo siguiente:

```
If x>23 gosub 500
…
499 rem por esta linea no se pasa y asi comento esta rutina
500 if x > 50 THEN ...
…
550 RETURN
```

Compactar en una línea todo aquello que sea logica de juego y pueda ser compactado. Por ejemplo:

```
10 if e1d=0 then e1x=e1x+1:if e1x>=70 then e1d=1:|SETUPSP,1,7,10
20 if e1d=1 then e1x=e1x-1:if e1x<=4 then e1d=0:|SETUPSP,1,7,9
```

cambiarlo por (fijate en el doble else para que aplique al primer if):

```
10 if e1d=0 then e1x=e1x+1:if e1x>=70 then e1d=1:|SETUPSP,1,7,10 else
else if e1d=1 then e1x=e1x-1:if e1x<=4 then e1d=0:|SETUPSP,1,7,9
```

Evitar ejecución de líneas de lógica innecesarias. Esta recomendación es la misma que la anterior pero con un estilo mas elegante de leer. Veamos un ejemplo. Con el ELSE 30 nos hemos evitado pasar por la línea 20 en muchos casos

En este ejemplo e1d es la dirección del sprite 1, e1x es su coordenada x

```
10 if e1d=0 then e1x=e1x+1:if e1x>=70 then e1d=1:|SETUPSP,1,7,10 else
30
20 if e1d=1 then e1x=e1x-1:if e1x<=4 then e1d=0:|SETUPSP,1,7,9
30 |LOCATESP,1,e1y,e1x
```

y ahora una versión aun mejor. Hemos eliminado un IF innecesario en la linea 20

```
10 if e1d=0 then e1x=e1x+1:if e1x>=70 then e1d=1:|SETUPSP,1,7,10 else
30
20 e1x=e1x-1:if e1x<=4 then e1d=0:|SETUPSP,1,7,9
30 |LOCATESP,1,e1y,e1x
```

En BASIC no sincronizar con el barrido de pantalla nunca ya que ralentiza el juego. Es decir, usar siempre |PRINTSPALL,1,0 en lugar de |PRINTSPALL,1,1. Si compilas el juego con algún compilador como "fabacom" entonces merece la pena sincronizar, tanto para fijar la velocidad de juego a 50 frames por segundo como para conseguir mayor suavidad en los movimientos
Gestiona el teclado (y en general esto es aplicable a cualquier cosa que hagas) ejecutando el menor numero de instrucciones. Aquí tienes un ejemplo (primero mal hecho y luego bien hecho), donde como mucho se pasa por 4 operaciones INKEYS con sus correspondientes IF. Ejecútalo mentalmente y comprobarás lo que digo. Es mucho mas rápida la segunda

Mal hecho (caso peor = 8 ejecuciones "IF INKEY")

```
1671 IF INKEY(27)=0 and INKEY(67)=0 THEN IF dir<>2 THEN
|SETUPSP,0,7,2:dir=2:goto 1746 ELSE |ANIMA,0:xn=xa+1:yn=ya-2:goto 1746

1672 IF INKEY(27)=0 and INKEY(69)=0 THEN IF dir<>8 THEN
|SETUPSP,0,7,8:dir=8:goto 1746 ELSE |ANIMA,0:xn=xa+1:yn=ya+2:goto 1746

1673 IF INKEY(34)=0 and INKEY(67)=0 THEN IF dir<>4 THEN
|SETUPSP,0,7,4:dir=4:goto 1746 ELSE |ANIMA,0:xn=xa-1:yn=ya-2:goto 1746

1674 IF INKEY(34)=0 and INKEY(69)=0 THEN IF dir<>6 THEN
|SETUPSP,0,7,6:dir=6:goto 1746 ELSE |ANIMA,0:xn=xa-1:yn=ya+2:goto 1746

1675 IF INKEY(27)=0 THEN IF dir<>1 THEN |SETUPSP,0,7,1:dir=1:goto 1746
ELSE |ANIMA,0:xn=xa+1:goto 1746
```

```
1676 IF INKEY(34)=0 THEN IF dir<>5 THEN |SETUPSP,0,7,5:dir=5:goto 1746
ELSE |ANIMA,0:xn=xa-1:goto 1746

1677 IF INKEY(67)=0 THEN IF dir<>3 THEN |SETUPSP,0,7,3:dir=3:goto 1746
ELSE |ANIMA,0:yn=ya-4:goto 1746

1678 IF INKEY(69)=0 THEN IF dir<>7 THEN |SETUPSP,0,7,7:dir=7:goto 1746
ELSE |ANIMA,0:yn=ya+4:goto 1746
```

Bien hecho (caso peor = 4 ejecuciones “IF INKEY”):

```
1510 if inkey(27)<>0 goto 1520

1511 if inkey(67)=0 then IF dir<>2 THEN |SETUPSP,0,7,2:dir=2:goto 1533
ELSE |ANIMA,0:xn=xa+1:yn=ya-2:goto 1533

1512 if inkey(69)=0 THEN IF dir<>8 THEN |SETUPSP,0,7,8:dir=8:goto 1533
ELSE |ANIMA,0:xn=xa+1:yn=ya+2:goto 1533

1513 IF dir<>1 THEN |SETUPSP,0,7,1:dir=1:goto 1533 ELSE
|ANIMA,0:xn=xa+1:goto 1533

1520 if inkey(34)<>0 goto 1530

1521 if INKEY(67)=0 THEN IF dir<>4 THEN |SETUPSP,0,7,4:dir=4:goto 1533
ELSE |ANIMA,0:xn=xa-1:yn=ya-2:goto 1533

1522 if INKEY(69)=0 THEN IF dir<>6 THEN |SETUPSP,0,7,6:dir=6:goto 1533
ELSE |ANIMA,0:xn=xa-1:yn=ya+2:goto 1533

1523 IF dir<>5 THEN |SETUPSP,0,7,5:dir=5:goto 1533 ELSE
|ANIMA,0:xn=xa-1:goto 1533

1530 IF INKEY(67)=0 THEN IF dir<>3 THEN |SETUPSP,0,7,3:dir=3:goto 1533
ELSE |ANIMA,0:yn=ya-4:goto 1533

1531 IF INKEY(69)=0 THEN IF dir<>7 THEN |SETUPSP,0,7,7:dir=7:goto 1533
ELSE |ANIMA,0:yn=ya+4:goto 1533

1532 return
```

En juegos donde la lógica de los enemigos requiere del uso del calculo de alguna función (como el coseno) , precalcula todo y guárdalo en un array que uses durante la ejecución de la lógica. Calcular durante la lógica del juego tiene un coste prohibitivo en BASIC

En juegos de naves es importante que no uses coordenadas negativas y si quieres que las naves aparezcan por los bordes y se perciba el clipping, reduce la pantalla un poco con SETLIMITS

En juegos de naves evita las comprobaciones con el layout. Esto supone casi 3.5ms de ahorro en cada ciclo de la lógica. Normalmente no requerirás de un layout en un juego de naves. Los escenarios los puedes simular con sprites que tengan desactivado el flag de colisión y que representen cráteres, bases espaciales, etc y junto con el suelo moteado con |STARS dará la sensación de tierra que se desplaza. Procura que tu nave sea el sprite 31, de este modo pasará por "encima" de los sprites que simulan ser el fondo, pues tu nave se imprimirá después

9.2 Aprovecha al máximo la memoria

Cada pantalla ocupa una memoria considerable, si se trata de un juego de pantallas definidas a través del layout. En un juego sencillo, cada pantalla ocupará un texto de BASIC de alrededor de 800 bytes y un código de unos 2000 bytes, por lo que es difícil hacer un juego de más de 10 pantallas (recuerda que dispones de 27KB para tu juego)

Una de las cosas fundamentales que debes hacer es reutilizar código de logica de enemigos en lugar de rescribirlo en cada pantalla.
La lógica de los enemigos de cada pantalla te puede ocupar más que el layout de la pantalla. En el juego "el mutante montoya", aproximadamente un tercio de cada pantalla es layout y los otros dos tercios son lógica. Si necesitas espacio para más pantallas lo más adecuado es "reutilizar" lógicas de enemigos entre pantallas, de un modo parametrizado. Por ejemplo un enemigo que se mueve de izquierda a derecha solo requiere 3 parámetros: donde comenzar (X, Y), donde termina su trayectoria por la derecha (X máxima) y donde termina por la izquierda (Xmin). Solo tendrías que inicializar estos 3 valores y con un GOSUB/RETURN ahorrarías líneas de BASIC en cada pantalla donde aparezca ese enemigo.

La clave es reutilizar código BASIC de unas pantallas en otras, igual que lo haces con la rutina de movimiento del personaje. Y dentro de cada pantalla si hay varios enemigos del mismo tipo, reutilizar el código de logica del enemigo, escribiéndolo sólo una vez.

Ejemplo: tres soldados paseándose por la pantalla

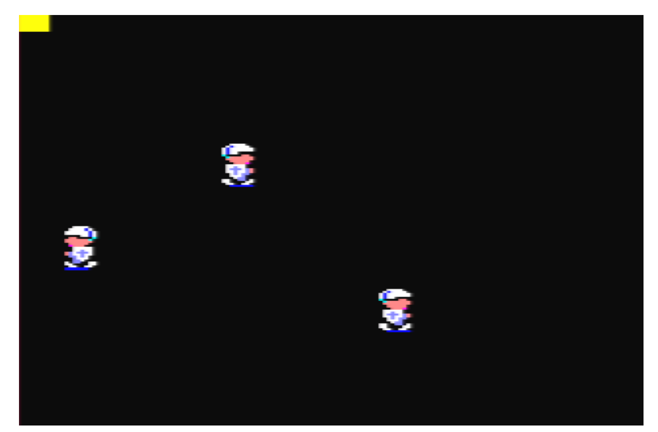


Fig. 14 tres soldados y una única rutina de lógica

```
10 MEMORY 26999
20 MODE 0: DEFINT A-Z: CALL &6B78:' install RSX
25 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT:'reset sprites
```

```
26 |SETLIMITS,0,80,0,200
30 'parametrizacion de 3 soldados
40 dim x(3)
50 x(1)=10:xmin(1)=10:xmax(1)=60:
y(1)=60:direccion(1)=0:|SETUPSP,1,7,9 :|SETUPSP,1,0,&x111
60 x(2)=20:xmin(1)=15:xmax(2)=40:
y(2)=100:direccion(2)=1:|SETUPSP,2,7,10 :|SETUPSP,2,0,&x111
70 x(3)=30:xmin(1)=5:xmax(3)=50:
y(3)=130:direccion(3)=0:|SETUPSP,3,7,9 :|SETUPSP,3,0,&x111
80 for i=1 to 3:|LOCATESP,i,y(i),x(i):next: 'colocamos los sprites

89 '----- BUCLE PRINCIPAL DEL JUEGO (CICLO DE JUEGO)----------------
90 for i=1 to 3:gosub 100:next:'llamada a los 3 soldados
91 |PRINTSPALL,1,0: ' anima e imprime los 3 soldados
95 goto 90
96 '------------- FIN DEL CICLO DE JUEGO ------------------------
99 '----- rutina de soldado -------
100 IF direccion(i)=0 THEN x(i)=x(i)+1:IF x(i)>=xmax(i) THEN
direccion(i)=1:|SETUPSP,i,7,10 ELSE 120
110 x(i)=x(i)-1:IF x(i)<=xmin(i) THEN direccion(i)=0:|SETUPSP,i,7,9
120 poke 27003+i*16, x(i):' le colocamos en la nueva coordenada
130 return
```

y como recomendaciones generales

Puedes superar las limitaciones de memoria mediante algoritmos que generen laberintos, o pantallas sin necesidad de almacenarlos. De este modo podrás hacer muchas mas pantallas. Esto requiere creatividad, desde luego, pero es posible.

Puedes reutilizar la lógica de enemigos de una pantalla en otra, ahorrando muchas líneas de código. Aprovecha el mecanismo GOSUB/RETURN para ello, definiendo lógicas comunes configurables por parámetros. Por ejemplo un guardián que se mueve de izquierda a derecha puedes situarlo en muchas pantallas a diferentes alturas y con diferentes limites en su trayectoria sin programarlo de nuevo.

También puedes hacer juegos que carguen por fases, de modo que no tengas todo el juego en memoria a la vez. Esto es un poco molesto para el usuario de cinta (CPC464) aunque no lo es para el de disco (CPC6128)

Intenta combinar layouts de pantallas consecutivas. Puede que las variaciones de una pantalla a la siguiente sean solo 10 líneas del layout (por ejemplo), ahorrando el 50% de la memoria para construirla (unos 400bytes de ahorro)

9.3 Técnica de "lógicas masivas" de sprites

A menudo vas a necesitar mover muchos sprites, sobre todo en juegos de arcade del espacio o de estilo "commando" (el clásico de capcom de 1985)

Para ello lo mas recomendable es hacer uso combinado de las funciones de movimiento automático y de movimiento relativo, que son |AUTOALL y |MOVERALL respectivamente. Podrías actuar por separado en las coordenadas de todos los sprites y actualizarlas usando POKE pero resultaría muy lento, inviable si quieres fluidez de movimientos.

La clave de lograr velocidad en muchos sprites es utilizar la técnica que he bautizado como "lógicas masivas". Esta técnica consiste fundamentalmente en ejecutar menos lógica en cada ciclo de juego (lo que se denomina "reducir la complejidad computacional") y para ello hay dos opciones:

Usar una sola lógica que afecta a muchos sprites a la vez (usando los flag de movimiento automatico y/o relativo)
Usar diferentes lógicas pero ejecutar solo una o unas pocas en cada ciclo del juego.

Ambas opciones tienen un mismo objetivo: ejecutar menos lógica en cada ciclo, permitiendo que todos los sprites se muevan a la vez pero tomando menos decisiones en cada ciclo del juego. La clave está en determinar qué logica o logicas ejecutar en cada ciclo. En el caso mas sencillo, si tenemos N sprites simplemente ejecutaremos una de las N logicas. Pero en casos mas complejos deberemos ser astutos para determinar que lógicas conviene ejecutar, como veremos al explicar como definir trayectorias complejas.

Vamos a ilustrar este concepto mediante un sencillo ejemplo en el siguiente apartado.

9.3.1 Ejemplo sencillo de lógica masiva

Volvamos al ejemplo de los 3 soldados. Esta vez vamos a ejecutar sólo la lógica de un soldado en cada ciclo de juego.
Para que a pesar de ello, la coordenada x de cada soldado siga avanzando, usaremos el flag de movimiento automático, en lugar de actualizarla nosotros.

```
10 MEMORY 26999
20 MODE 0: DEFINT A-Z: CALL &6B78:' install RSX
25 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT:'reset sprites
26 |SETLIMITS,0,80,0,200
30 'parametrizacion de 3 soldados
40 dim x(3):x%=0
50 x(1)=10:xmin(1)=10:xmax(1)=60:
y(1)=60:direccion(1)=0:|SETUPSP,1,7,9 :|SETUPSP,1,0,&x1111:
|SETUPSP,1,5,0: |SETUPSP,1,6,1
60 x(2)=20:xmin(1)=15:xmax(2)=40:
y(2)=100:direccion(2)=1:|SETUPSP,2,7,10 :|SETUPSP,2,0,&x1111:
|SETUPSP,2,5,0: |SETUPSP,2,6,-1
70 x(3)=30:xmin(1)=5:xmax(3)=50:
y(3)=130:direccion(3)=0:|SETUPSP,3,7,9 :|SETUPSP,3,0,&x1111:
|SETUPSP,3,5,0: |SETUPSP,3,6,1
80 for i=1 to 3:|LOCATESP,i,y(i),x(i):next: 'colocamos los sprites
81 i=0
```

```
89 '----- BUCLE PRINCIPAL DEL JUEGO (CICLO DE JUEGO)----------------
90 i=i+1:gosub 100
92 if i=3 then i=0
93 |AUTOALL
94 |PRINTSPALL,1,0: ' anima e imprime los 3 soldados
95 goto 90
96 '------------- FIN DEL CICLO DE JUEGO ------------------------

99  '----- rutina de soldado -------
100 |PEEK,27003+i*16,@x%: x(i)=x%
101 IF direccion(i)=0 THEN IF x(i)>=xmax(i) THEN
direccion(i)=1:|SETUPSP,i,7,10: |SETUPSP,i,6,-1 ELSE return
110 IF x(i)<=xmin(i) THEN direccion(i)=0:|SETUPSP,i,7,9 :
|SETUPSP,i,6,1
120 return
```

Pruébalo y comprobarás que se ha multiplicando la velocidad casi por 3. cada soldado tiene su propia lógica, pero solo ejecutamos una en cada ciclo de juego, aligerando muchísimo el ciclo de juego.

La única limitación es que al ejecutar la lógica de cada soldado una de cada 3 veces, la coordenada podría sobrepasar el limite que hemos establecido durante dos ciclos. Eso hace que debamos ser mas cuidadosos al fijar el límite, asegurándonos al ejecutarlo que nunca invade y borra un muro de nuestro laberinto de pantalla, por ejemplo.

Voy a tratar de explicar este problema con más precisión:
Supongamos que tenemos 8 sprites y nuestro sprite se mueve en todos los ciclos pero solo ejecutamos su lógica una de cada 8 veces.

Imagínate un sprite que está en la posición x=20 y queremos que se mueva hasta la posición x=30 y dar la vuelta. Consideremos que el sprite tiene un movimiento automático con Vx=1.
En ese caso comprobaremos su posición cuando x=20, x=28 , x=36. al llegar a 36 nos daremos cuenta de que nos hemos pasado!!! y cambiaremos la velocidad del sprite a Vx=-1

Como ves el control de los limites de la trayectoria no es preciso , a menos que tengamos en cuenta esta circunstancia y fijemos el limite en algo que podamos controlar, que será Xfinal = Xinicial + n*8.

Esta limitación es minúscula si la comparamos con la ventaja de mover muchos sprites a gran velocidad. Con algo de astucia podemos incluso ejecutar la lógica menos veces, de modo que solo uno de cada dos ciclos se ejecute algún tipo de lógica de enemigos.

9.3.2 Mueve 32 sprites con lógicas masivas

Ahora vamos a ver un sencillo ejemplo para mover 32 sprites simultáneamente y suavemente (a 14fps) . Es perfectamente posible. Solo un fantasma va a tomar decisiones en cada ciclo, aunque se van a mover todos los fantasmas en todos los ciclos. También podemos animar a todos (asociándoles una secuencia de animación y usando

|PRINTSPALL,1,0) y seguirá quedando suave, pero aun parecerá que hay mayor movimiento pues el aleteo de las alas de una mosca (por ejemplo) genera mucha sensación de movimiento

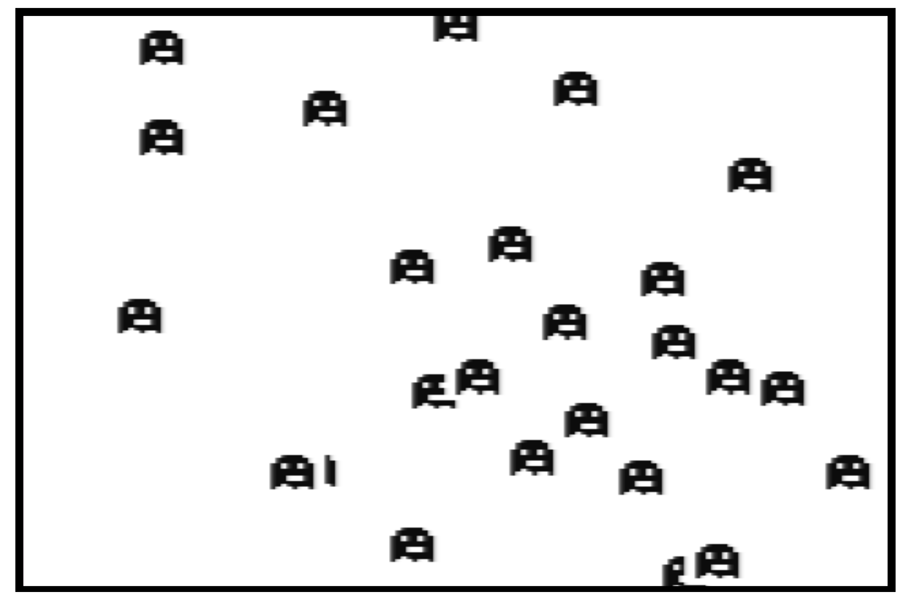


Fig. 15 con lógicas masivas puedes mover 32 sprites simultáneamente

Lo que hemos hecho ha sido reducir la complejidad computacional. Hemos partido de un problema de "orden N", siendo N el número de sprites. Suponiendo que cada lógica de sprite requiera 3 instrucciones BASIC, en principio habría que ejecutar N x 3 instrucciones en cada ciclo. Con la técnica de "lógicas masivas", transformamos el problema de "orden N" en un problema de "orden 1". Se llama problemas de "orden 1" a los que involucran un número constante de operaciones independientemente del tamaño del problema. En este caso hemos pasado de Nx3 operaciones BASIC a sólo 3 operaciones BASIC. Esta reducción de complejidad es la clave del alto rendimiento de la técnica de lógicas masivas.

```
1 MODE 0
10 MEMORY 26999: CALL &6B78
20 DEFINT a-z
25' reset enemigos
30 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT
35 ' num enemigos de 12 x 16 (6bytes de ancho x 16 lineas)
36 num=32: x%=0:y%=0
40 FOR i=0 TO num-1:|SETUPSP,i,9,&8ee2: |SETUPSP,i,0,&X1111:
41 |LOCATESP,i,rnd*200,rnd*80
42 next
43 i=0
45 gosub 100
46 i=i+1: if i=num then i=0
50 |PRINTSPALL,0,0
60 |AUTOALL
70 goto 45

100 |peek,27001+i*16,@y%
```

```
110 |peek,27003+i*16,@x%
120 if y%<=0 then |SETUPSP,i,5,2:|SETUPSP,i,6,0: return
130 if y%>=190 then |SETUPSP,i,5,-2:|SETUPSP,i,6,0: return
140 if x%<=0 then |SETUPSP,i,5,0:|SETUPSP,i,6,1: return
150 if x%>=76 then |SETUPSP,i,5,0:|SETUPSP,i,6,-1: return

160 azar=rnd*3
170 if azar=0 then |SETUPSP,i,5,2: |SETUPSP,i,6,0:return
180 if azar=1 then |SETUPSP,i,5,-2:|SETUPSP,i,6,0:return
190 if azar=2 then |SETUPSP,i,5,0:|SETUPSP,i,6,1:return
200 if azar=3 then |SETUPSP,i,5,0:|SETUPSP,i,6,-1:return
```

9.3.3 Técnica de lógicas masivas en juegos tipo “pacman”

Si tienes muchos enemigos y deben tomar decisiones en cada bifurcación de un laberinto, quizás pienses que la técnica de lógica masiva no es precisa pues cada enemigo no comprueba su posición en cada ciclo de juego, pero esto se puede solucionar con un sencillo “truco”. Es simplemente colocar a los enemigos en posiciones bien escogidas al empezar el juego.

Supongamos que tienes 8 enemigos y que las bifurcaciones del laberinto ocurren en múltiplos de 8.

Si el primer enemigo está en una posición múltiplo de 8 le tocará ejecutar su lógica. Al segundo enemigo le toca ejecutar su lógica de decisión en el ciclo siguiente. Si no se encuentra en una posición de bifurcación del laberinto no podrá cambiar su rumbo

Para que “encaje” su posición con un múltiplo de 8 y asi poder decidir que camino tomar en la bifurcación, simplemente empezamos el juego con este segundo enemigo colocado en un múltiplo de 8 menos uno. Considerando coordenadas que comienzan en cero, los múltiplos de 8 son:

Primer enemigo: posición 0 o 8 o 16 o 24 o 32 o XX (en eje x o y, da igual)
Segundo enemigo: posición 7 o 15 o 23 ...
Tercer enemigo : posición 6 o 14 o 22 ...

Y asi sucesivamente. Colocas a tus enemigos siguiendo esta regla:

Posición = múltiplo de 8 – n, siendo n el numero de sprite

Y cada vez que le toque a un enemigo ejecutar su lógica, podrá encontrarse en una bifurcación. Eso no significa que no deba comprobar que no se encuentra en una posición en mitad de un pasillo sin bifurcaciones. Debe comprobarlo y para ello puede usar PEEK contra un carácter del layout, ya que su coordenada dividida entre 8 es precisamente una posición del layout. El PEEK es muy rápido (unos 0.9 ms) frente a la detección de colisión que consume mas de 3ms. Con PEEK compruebas si la posición superior esta ocupada por un ladrillo (por ejemplo) y si hay via libre entonces el enemigo podría tomar esa nueva dirección.

¿ y si solo tienes 5 enemigos? Pues muy fácil. Hay ciclos que puedes simplemente no ejecutar ninguna lógica y así los cinco siempre toman decisiones al encontrarse en posiciones de bifurcación, múltiplos de 8.

9.3.4 Movimiento “en bloque” de escuadrones

Si lo que quieres es simplemente mover a la vez un escuadrón en una dirección, cualquiera de las dos funciones siguientes de la librería 8BP te servirán:

- Si usas |AUTOALL tienes que ponerle velocidad automática a los sprites en la dirección que quieras (en Vx, en Vy o en ambas) y por supuesto activar el bit 4 del byte de estado. El comando AUTOALL no tiene parámetros

- Si usas |MOVERALL tienes que activar el bit 5 del byte de estado a los sprites que vayas a mover. Este comando requiere que como parámetros introduzcas cuanto movimiento relativo en Y y en X deseas

El uso combinado de ambas estrategias te puede permitir mover dos escuadrones con trayectorias complejas y diferentes entre si, veamos un ejemplo

En este videojuego hemos definido el escuadrón de la derecha con un movimiento relativo (usando MOVERALL), siguiendo una trayectoria circular almacenada en un array. El grupo de la izquierda tiene el flag de movimiento relativo desactivado pero tiene activado el flag de movimiento automático, y usando AUTOALL se desplazan hacia abajo pues todos ellos tienen Vy=2

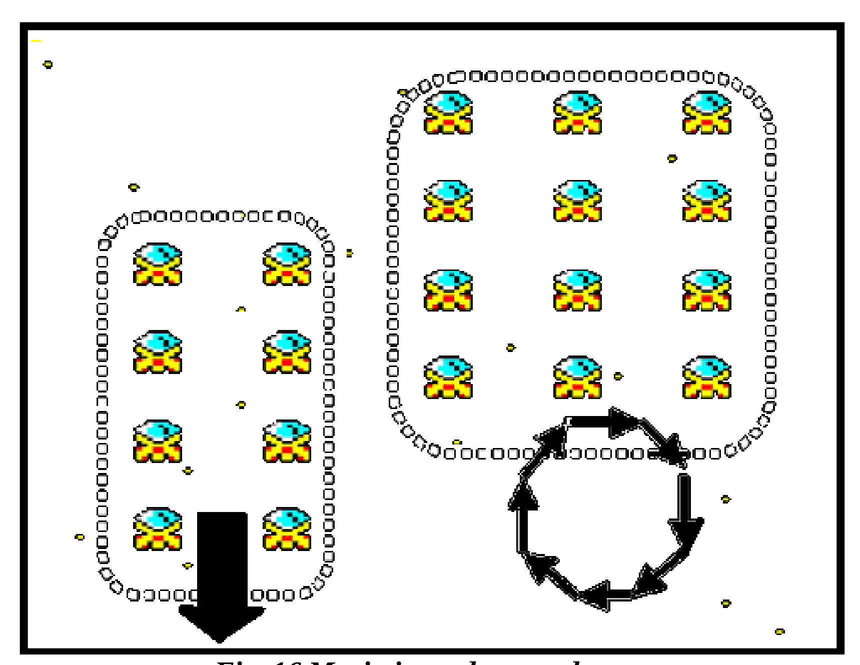


Fig. 16 Movimiento de escuadrones

aquí tienes el listado del ejemplo

```
1 MODE 0
10 MEMORY 26999
```

```
20 DEFINT a-z
25 REM la ruta de movimiento relativo la almacenamos en los arrays ry,
rx
30 DIM rx(24):DIM ry(24)
40 rx(0)=1:ry(0)=2
50 rx(1)=1:ry(1)=2
60 rx(2)=1:ry(2)=2
70 rx(3)=0:ry(3)=2
80 rx(4)=0:ry(4)=2
90 rx(5)=0:ry(5)=2
100 rx(6)=-1:ry(6)=2
110 rx(7)=-1:ry(7)=2
120 rx(8)=-1:ry(8)=2
130 rx(9)=-1:ry(9)=0
140 rx(10)=-1:ry(10)=0
150 rx(11)=-1:ry(11)=0
160 rx(12)=-1:ry(12)=-2
170 rx(13)=-1:ry(13)=-2
180 rx(14)=-1:ry(14)=-2
190 rx(15)=0:ry(15)=-2
200 rx(16)=0:ry(16)=-2
201 rx(17)=0:ry(17)=-2
202 rx(18)=1:ry(18)=-2
203 rx(19)=1:ry(19)=-2
204 rx(20)=1:ry(20)=-2
205 rx(21)=1:ry(21)=0
206 rx(22)=1:ry(22)=0
207 rx(23)=1:ry(23)=0
210 rem ------------inicializamos los escuadrones -------------
220 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT
230 FOR j=0 TO 16 STEP 4
240 FOR i=j TO j+3
250 |SETUPSP,i,0,&X10111:|SETUPSP,i,7,14 : |SETUPSP,i,6,1
:|SETUPSP,i,5,1
260 |LOCATESP,i,10+(i-j)*28,3*j+10
270 NEXT
280 NEXT j
281 rem inicializamos el segundo escuadron con movimiento automatico
282 for i=0 to 7 : |SETUPSP,i,0,&X01111: |SETUPSP,i,5,2:
|SETUPSP,i,6,0:next

310 rem -----bucle de logica del programa -------------------
420 |PRINTSPALL,0,0
421 |AUTOALL
430 |MOVERALL,ry(t),rx(t)
431 |STARS,0,20,1,3,0
432 rem aqui cambiamos el indice de la ruta
440 t=t+1 : IF t=24 THEN t=0
470 GOTO 420
```

En el ejemplo anterior todas las naves se mueven “en bloque”. En cualquier momento una de ellas puede desactivar su flag de movimiento relativo y/o automatico y tomar

vida propia, con una lógica específica individual que le haga volar hacia nuestra nave y atacarnos.

Pero si lo que deseas es algo mas avanzado, como hacer que un grupo de naves no se muevan “en bloque” sino que sigan una trayectoria en fila “india”, de modo que la nave que está en cabeza hace unos movimientos que van imitando las demás, puedes usar estrategias como la que te voy a describir a continuación.

9.3.5 Lógicas masivas: Movimiento “en fila india”

Un grupo de naves pueden tener un movimiento automático horizontal pero tienen una lógica de modo que cuando su coordenada x exceda de un cierto valor, pasan a tener movimiento automático y relativo a la vez. Y cuando pasan de otra coordenada x vuelven a desactivar el movimiento relativo

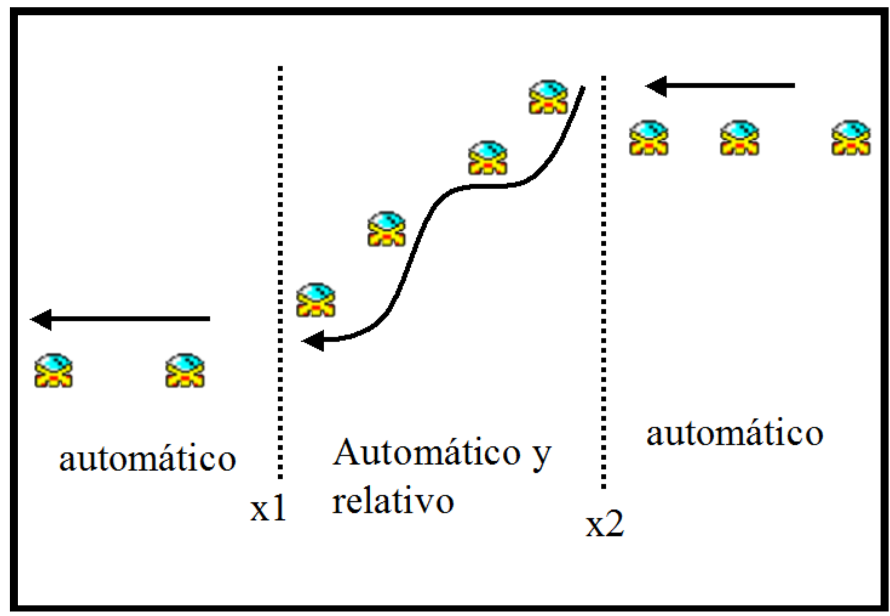


Fig. 17 Trayectorias en fila india

Esta sencilla estrategia te permite hacer el efecto de “fila india” de naves de un modo muy eficiente, sin actuar sobre las coordenadas de ninguna de forma individual.

La forma de hacerlo no es controlando la coordenada x de cada nave pues eso implicaría un bucle de comprobaciones muy lento. La forma rápida es mediante un contador de tiempo. En cada ciclo de juego se incrementa. Algo como lo que te muestro a continuación

```
t=t+1: if t>= 20 and t<= 25 then |SETUPSP,t-20,0,&x10111
```

Con esa línea dentro de la lógica del juego, cada vez que se ejecuta se incrementa el contador y cuando llega a 25, una a una, las naves correspondientes a los sprites 0,1,2,3,4 van cambiando de modo de funcionamiento. Cuando el contador llega a 30, habrán pasado las 5 naves al nuevo modo de comportamiento. El contador hace las veces de control de la coordenada x de todas las naves pero de un modo infinitamente más eficaz. Esta estrategia es en esencia no ejecutar la lógica de cada nave en cada ciclo, sino tan solo la lógica de la nave que interesa ejecutar en cada momento. Imagina

lo costoso que sería hacer la comprobación de la coordenada x en 8 naves cada ciclo. De acuerdo a la tabla de rendimiento, un IF se toma 1.42ms y por lo tanto 8 naves como mínimo se tomarían unos 12 ms para hacer lo mismo que hemos hecho con un solo IF aplicado al contador de tiempo.

9.3.6 Lógicas masivas: Trayectorias complejas

Si lo que deseas es algo aun mas complejo, donde las naves describan un circulo en fila india, deberás basarte sólo en movimiento automático. Piensa en la siguiente figura

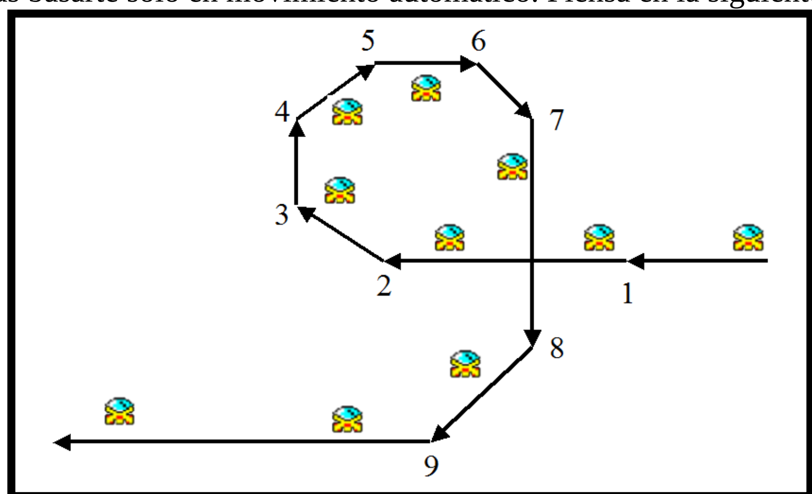


Fig. 18 Puntos de control de las trayectorias

Con un solo contador y nueve puntos de control puedes ir cambiando una a una las Vy,Vx de cada nave, de modo que en cada ciclo de tu lógica solo se actúa como mucho sobre una nave

Por ejemplo el punto de control “2” sería algo como

```
t=t+1:if t>= 15 and t<=20 then |SETUPSP,t-15,5,-1:|SETUPSP,t-15,5,-1
```

establecer 9 puntos de control para el contador supone establecer 9 sentencias IF por las que pasa el programa principal pero puedes utilizar estrategias para saltar con GOTO, tales como

```
if t <50 goto primer grupo de IF
if t <100 goto segundo grupo
if t <100 goto tercer grupo
etc
```

de esta manera con solo 3 IF en cada ciclo del juego podrás controlar los 9 puntos de control.

Vamos a ver un ejemplo simplificado. El siguiente ejemplo funciona muy suave con un escuadrón de 8 naves. Tiene sólo 2 puntos de control pero es esencialmente la misma estrategia. He utilizado dos contadores para evitar que las naves cambien de modo de funcionamiento demasiado seguido. El contador que las hace cambiar es “t” y solo

avanza cada vez que el contador “tt” cuenta hasta 10, momento en el que “tt” vuelve a comenzar. El número 10 es precisamente la separación entre sprites considerando la coordenada x.

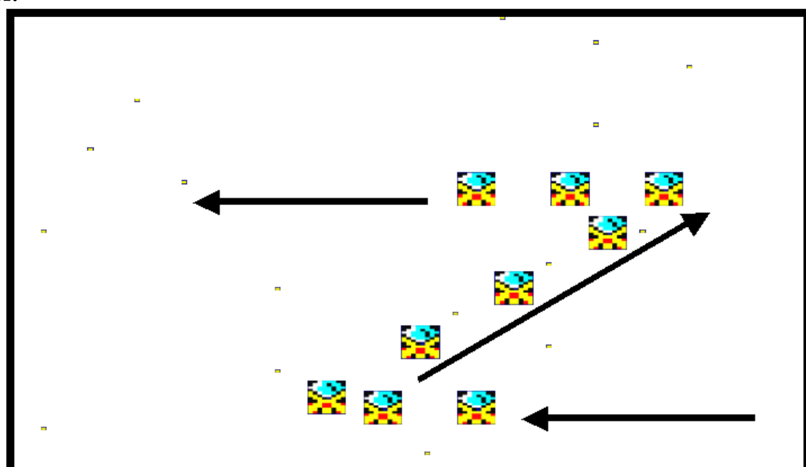


Fig. 19 Trayectorias complejas con lógicas masivas

```
1 MODE 0
10 MEMORY 26999
20 DEFINT a-z
100 rem inicializacion--------------------------
220 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT
221 FOR j=0 TO 7:
222 |SETUPSP,j,0,&X1111:|setupsp,j,7,14:
223 |setupsp,j,5,0:|setupsp,j,6,-1:
224 |locatesp,j,150,40+j*10:
225 NEXT

400 t=0:tt=0:rem bucle de logica juego----------------
420 |PRINTSPALL,0,0
421 |autoall
431 |STARS,0,20,1,0,-2
432 tt=tt+1: if tt=10 then tt=0:gosub 1000
441 rem primer tramo (sin logica)
442 goto 420

499 rem segundo tramo------------------
500 |setupsp,t-1,5,-2:|setupsp,t-1,6,1:
510 return
599 rem tramo 3-----------------------
600 |setupsp,t-5,5,0:|setupsp,t-5,6,-1:
610 return
999 rutina de contador c-----------------
1000 t=t+1:if t<=8 then gosub 500
1010 if t >4 and t<=13 then gosub 600
1020 return
```

Vamos a ver otro ejemplo para clarificar aun mas el concepto de utilizar un contador de tiempo como mecanismo para alterar los comportamientos de sprites, sin que haya que ejecutar lógica independiente para cada uno.

En este ejemplo los 8 enemigos empiezan dispuestos en dos diagonales de 4 cada una. Cada grupo de 4 avanza en dirección contraria y a medida que llegan al final cambian de dirección, produciendo un efecto de entrelazamiento de trayectorias muy atractivo

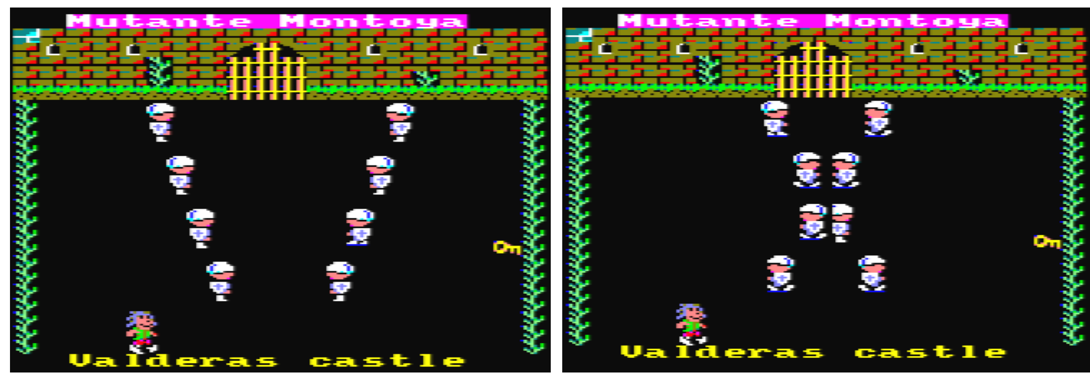


Fig. 20 uso de Logicas masivas en el "Mutante Montoya"

En esta lógica, cada 4 ciclos del bucle principal (controlados por el contador cc) se ejecuta el incremento del contador c. Y cuando c se encuentra entre 10 y 15 (ciclos 40 a 60) se van alterando el comportamiento de los soldados que se encuentran colocados en diagonales opuestas. Solo se altera el comportamiento de 2 soldados a la vez. Transcurrido un tiempo correspondiente a c=20 (80 unidades de tiempo) se empiezan a dar la vuelta otra vez.

El bucle principal controla también la colisión entre sprites y la salida de la pantalla cuando la coordenada Y del personaje (controlada por "yn") es menor que 26

```
4401 c1=10:c2=20:c=0:cc=0
4409 '----- bucle de logica principal ---------
4410 GOSUB 1500: ' control del personaje
4421 cc=cc+1: if cc=4 then cc=0: gosub 4700
4609 |AUTOALL : ' movimiento automatico de 8 sprites
4610 |PRINTSPALL,1,0 'impresion masiva de sprites (9= 8+personaje)
4611 |COLSP,0,@cs%:IF cs%<32 THEN GOSUB 600:GOTO 4020
4612 IF yn<26 THEN RETURN
4620 GOTO 4410
4699 '----- rutinas de control del escuadron ----
4700 c=c+1:if c>c1 and c<=c1+4 then gosub 4800
4701 if c>c2 and c<=c2+4 then gosub 4900
4702 if c=30 then c=10
4710 return
4799 '----- a darse la vuelta de 2 en 2 --------
4800 |setupsp,c1+5-c,7,10:|setupsp,c1+5-c,6,-1
4801 |setupsp,c1+5-c+4,7,9:|setupsp,c1+5-c+4,6,1
4810 return
4899 '----- a darse la vuelta otra vez --------
4900 |setupsp,c2+5-c,7,9:|setupsp,c2+5-c,6,1
4901 |setupsp,c2+5-c+4,7,10:|setupsp,c2+5-c+4,6,-1
4910 return
```

10 Juegos con scroll

La librería 8bp no puede hacer scroll...pero puede hacer que parezca que hay un scroll, de modo que es como si pudieses hacerlo. Vamos a ver varios ejemplos

10.1 Scroll de estrellas o tierra moteada

En la librería 8BP dispones de una función muy sencilla de utilizar para crear un efecto de fondo de estrellas que se mueven, dando la sensación de scroll. Se trata de la funcion |STARS.
Esta funcion es capaz de mover hasta 20 estrellas simultáneamente sin alterar tus sprites, de modo que es como si pasasen "por debajo"

|STARS,<banco>,<num estrellas>,<color>,<dy>,<dx>

Dispones de dos bancos de estrellas de modo que puedes combinar dos comandos STARS para trabajar con ambos bancos y asi desplazar estrellas a distinta velocidad, dando sensación de profundidad.

Cada banco consiste en 20 pares de bytes representando coordenadas (y,x) .

Banco 0: desde la dirección 42540 hasta 42579 (son 40 bytes)
Banco 1: desde la dirección 42580 hasta 42619 (son 40 bytes)

Una forma de generar estrellas aleatorias en el banco 1 sería:

FOR dir=42580 TO 42618 STEP 2: POKE dir,RND*200:POKE dir+1,RND*80:NEXT

Para una descripción detallada del comando, consulta el capitulo de "guia de referencia". En ese capítulo encontraras distintos ejemplos para simular estrellas, tierra, estrellas con dos planos de profundidad, lluvia o incluso nieve. Probablemente con imaginación sea posible simular mas cosas con esta misma funcion.

Por ejemplo si colocas las estrellas en secuencias de 2 o tres pixeles en diagonal, en lugar de repartirlas aleatoriamente, podras conseguir un desplazamiento de movimiento de "segmentos", algo que podría ser ideal para simular lluvia.

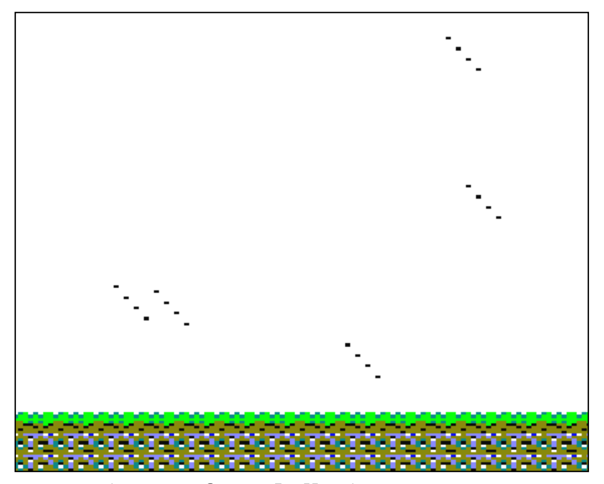


Fig. 21 Efecto de lluvia con STARS

```
10 MEMORY 26999
20 'pongo estrellas aleatorias
21 banco1=42580
30 FOR dir=banco1 TO banco1+40 STEP 8:
40 y=INT(RND*190+10):x=INT(RND*60+10)

60 POKE dir,y:POKE dir+1,x:
70 POKE dir+2,(y+4):POKE dir+3,x+1
80 POKE dir+4,(y+8):POKE dir+5,x+2
90 POKE dir+6,(y+12):POKE dir+7,x+3
100 NEXT
110 MODE 0: DEFINT A-Z: CALL &6B78:' install RSX
120 CALL &BC02:'restaura paleta por defecto por si acaso
130 INK 0,0
140 |SETLIMITS,0,80,50,200: ' establecemos los limites de la pantalla
de juego

141 cadena$="YYYYYYYYYYYYYYYYYYYY"

142 |LAYOUT,22,0,@cadena$
143 cadena$="PPPPPPPPPPPPPPPPPPPP"
144 |LAYOUT,23,0,@cadena$
145 |LAYOUT,24,0,@cadena$
150 '----- ciclo de juego--------
160 |STARS,1,20,4,2,1
170 GOTO 160
```

Como el ejemplo de doble plano de estrellas lo tienes en el capitulo de referencia de la librería, aquí vamos a ver un ejemplo en el que una nave espacial sobrevuela un planeta de tierra moteada, con sensación de scroll vertical



En este ejemplo trata de apreciar como las estrellas (o motas del suelo) pasan literalmente por debajo de la nave, sin afectarla

Fig. 22 Efecto de tierra moteada con STARS

```
10 MEMORY 26999
11 'pongo estrellas aleatorias
12 FOR dir=42580 TO 42618 STEP 2: POKE dir,RND*200:POKE
dir+1,RND*80:NEXT
20 MODE 0: DEFINT A-Z: CALL &6B78:' install RSX
25 call &bc02:'restaura paleta por defecto por si acaso
26 ink 0,13:'fondo gris
30 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT:'reset sprites
40 |SETLIMITS,12,80,0,186: ' establecemos los limites de la pantalla
de juego

41' vamos a crear una nave en el sprite 31
42 |SETUPSP,31,0,&1:' status
43 nave = &a2f8: |SETUPSP,31,9,nave:' asigno imagen al sprite 31
44 x=40:y=150: ' coordenadas de nave

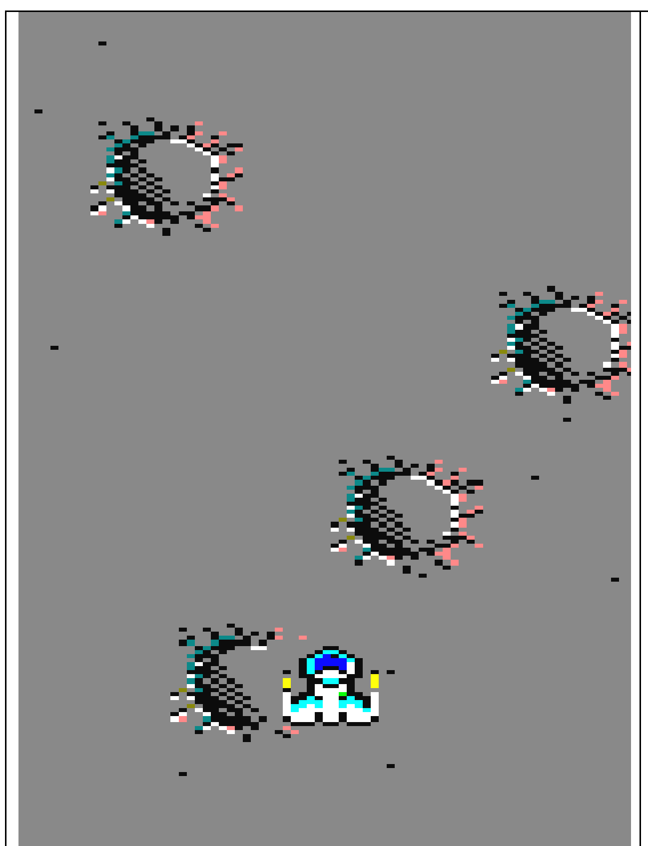
49'----- ciclo de juego--------
50 |STARS,1,20,5,1,0:' estrellas negras es como un suelo
55 gosub 100:' movimiento de la nave
60 |PRINTSPALL,0,0
70 goto 50


99 ' rutina movimiento nave ------------
100 IF INKEY(27)=0 THEN x=x+1:GOTO 120
110 IF INKEY(34)=0 THEN x=x-1
120 |LOCATESP,31,y,x
130 RETURN
```

10.2 Cráteres en la luna

Ahora haciendo uso combinado del movimiento relativo, del scroll de estrellas y del orden en que se imprimen los sprites vamos a ver un ejemplo de cómo simular que una nave sobrevuela la luna, es decir, como simular un scroll de pantalla con estos elementos

Primeramente hemos escogido el sprite 31 para nuestra nave, porque eso hará que se imprima la última. Los sprites se imprimen en orden, comenzando por el cero y acabando en el 31. Si un crater es un sprite inferior a 31 se imprimirá antes que la nave y la nave quedará "por encima", dando la sensación de que lo esta sobrevolando.



En la imagen aparece la nave por encima de un cráter. Como el fondo de la nave y el suelo son del mismo color, queda “aceptable” aunque no sea una sobreescritura

Es muy importante que las estrellas (motas del suelo) y los cráteres se muevan a la misma velocidad. Para ello tanto el comando |MOVERALL como el comando |STARS deben tener el mismo valor del parámetro dy

Fig. 23 sobrevolando la luna

Este es el código BASIC

```
10 MEMORY 26999
11 'pongo estrellas aleatorias
12 FOR dir=42580 TO 42618 STEP 2: POKE dir,RND*200:POKE
dir+1,RND*80:NEXT
20 MODE 0: DEFINT A-Z: CALL &6B78:' install RSX
25 call &bc02:'restaura paleta por defecto por si acaso
26 ink 0,13:'fondo gris
30 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT:'reset sprites
40 |SETLIMITS,12,80,0,186: ' establecemos los limites de la pantalla
de juego

41' vamos a crear una nave en el sprite 31
42 |SETUPSP,31,0,&1:' status
43 nave = &a2f8: |SETUPSP,31,9,nave:' asigno imagen al sprite 31
45 x=40:y=150: ' coordenadas de nave

46' ahora los crateres
47 crater=&a39a: cy%=0
48 for i=0 to 3 : |SETUPSP,i,9,crater:
49 |SETUPSP,i,0,&x10001: ' impresion y movimiento relativo
50 x(i)=rnd*40+20:y(i)=i*40
60 |locatesp,i,y(i),x(i)
```

```
70 next
71 t=0

80'----- ciclo de juego--------
81 |STARS,1,20,5,3,0:' movimiento estrellas negras
82 gosub 100:' movimiento de la nave
83 |MOVERALL,3,0: 'movimiento de crateres
84 t=t+1: if t> 10 then t=0:gosub 200:' control de crateres
90 |PRINTSPALL,0,0:' impresion de nave y crateres
91 goto 81


99 ' rutina movimiento nave -------------
100 IF INKEY(27)=0 THEN x=x+1:GOTO 120
110 IF INKEY(34)=0 THEN x=x-1
120 |LOCATESP,31,y,x
130 RETURN

199' control de reentrada de crateres
200 c=c+1
210 if c=6 then c=0
220 |PEEK,27001+c*16,@cy%
230 if cy%>200 then |POKE,27001+c*16,-20
240 return
```

En el siguiente ejemplo se ha puesto un escuadron de naves espaciales, los crateres y tu nave espacial (9 sprites en total y las estrellas). Gracias a los comandos MOVERALL y AUTOALL, es posible mover todo esto a un ritmo adecuado para un juego de arcade

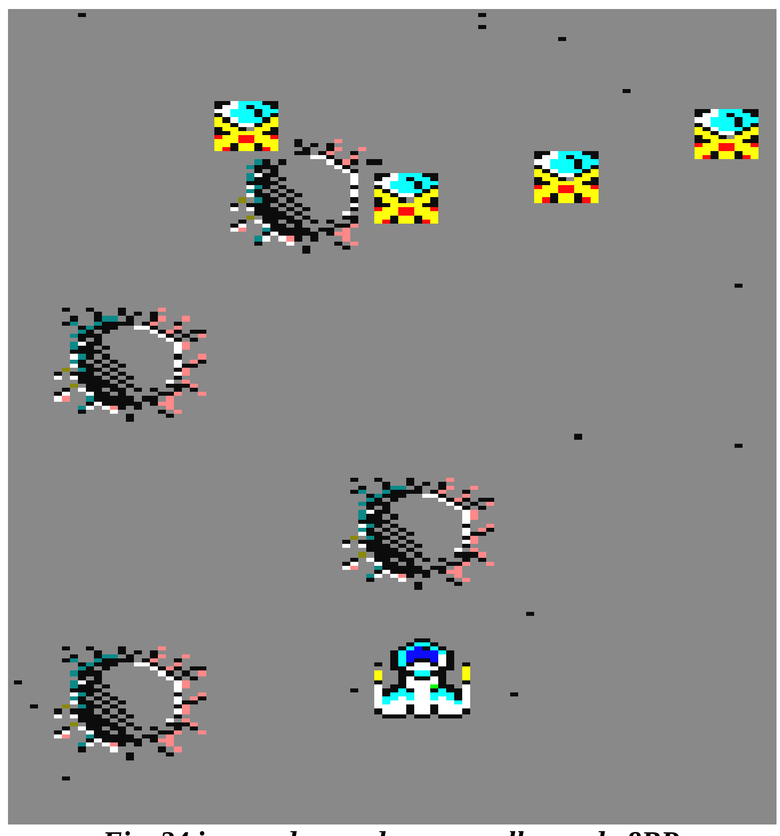


Fig. 24 juegos de arcade con scroll usando 8BP

```
10 MEMORY 26999
11 'pongo estrellas aleatorias
12 FOR dir=42580 TO 42618 STEP 2: POKE dir,RND*200:POKE
dir+1,RND*80:NEXT
20 MODE 0: DEFINT A-Z: CALL &6B78:' install RSX
25 CALL &BC02:'restaura paleta por defecto por si acaso
26 INK 0,13:'fondo gris
30 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT:'reset sprites
40 |SETLIMITS,12,80,0,186: ' establecemos los limites de la pantalla
de juego
41 ' vamos a crear una nave en el sprite 31
42 |SETUPSP,31,0,&1:' status
43 nave = &A2F8: |SETUPSP,31,9,nave:' asigno imagen al sprite 31
45 x=40:y=150: ' coordenadas de nave
46 ' ahora los crateres
47 crater=&A39A: cy%=0
48 FOR i=0 TO 3 : |SETUPSP,i,9,crater:
49 |SETUPSP,i,0,&X1001:|SETUPSP,i,5,3:|SETUPSP,i,6,0:  ' impresion y
movimiento auto
50 x(i)=RND*40+20:y(i)=i*40
60 |LOCATESP,i,y(i),x(i)
70 NEXT
71 t=0
75 ' naves
76 FOR i=4 TO 7 :
|SETUPSP,i,7,14:|SETUPSP,i,0,&X10101:|LOCATESP,i,RND*20,i*10-20
77 NEXT :
78 inc=1
80 '----- ciclo de juego--------
81 |STARS,1,20,5,3,0:' movimiento estrellas negras
82 GOSUB 100:' movimiento de la nave
84 t=t+1: IF t> 10 THEN inc=-inc: t=0:GOSUB 200:' control de crateres
85 |MOVERALL,2,inc
86 |AUTOALL
90 |PRINTSPALL,0,0
91 GOTO 81
99 ' rutina movimiento nave -------------
100 IF INKEY(27)=0 THEN x=x+1:GOTO 120
110 IF INKEY(34)=0 THEN x=x-1
120 |LOCATESP,31,y,x
130 RETURN
199 ' control de reentrada de crateres
200 c=c+1
210 IF c=4 THEN c=0
220 |PEEK,27001+c*16,@cy%
230 IF cy%>200 THEN |POKE,27001+c*16,-20
240 RETURN
```

10.3 Montañas y lagos

La técnica para pintar montañas en juegos con scroll horizontal y lagos en juegos con scroll vertical es la misma

Lo que haremos es pintar solo el comienzo de la montaña, mediante un sprite que nos sirve para pintar su lado derecho. Pondremos tantos como queramos. En este caso yo he puesto tres

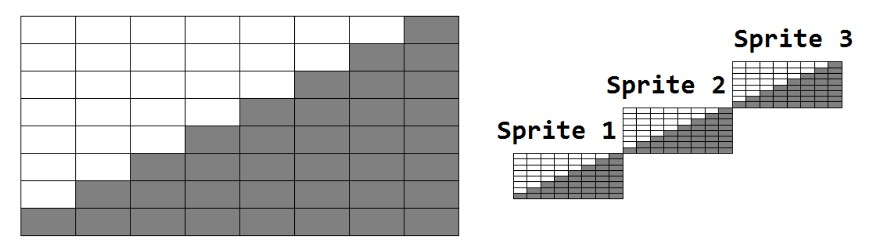


Fig. 25definir la ladera de una montaña con varios sprites

hacemos lo mismo con una imagen espejada que asociaremos a otros 3 sprites, y los situaremos a la derecha, construyendo el lateral izquierdo de la montaña. Cuidado de que la imagen espejada al menos tenga la ultima columna de pixels a cero. Esto le permitirá borrarse a si misma al avanzar a la izquierda.

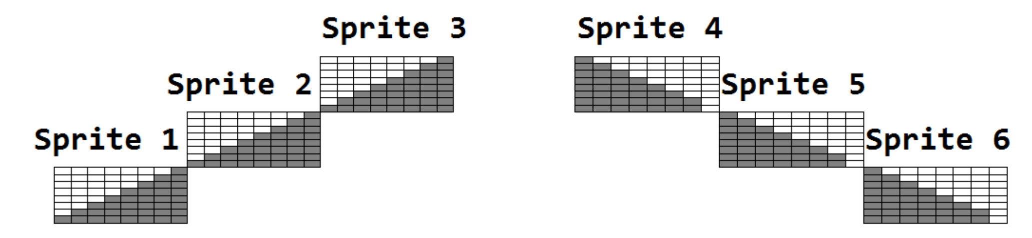


Fig. 26 disposición de Sprites para construir una montaña

Al mover todos los sprites hacia la izquierda mediante movimiento automatico o relativo, los sprites de la izquierda empezarán a “manchar” el fondo y por consiguiente “rellenando” la montaña, al tiempo los sprites de la derecha empezarán a limpiarlo. Si la montaña aparece poco a poco entrando en la pantalla, parecerá un sprite de una montaña enorme, cuando en realidad se trata de 6 pequeños sprites

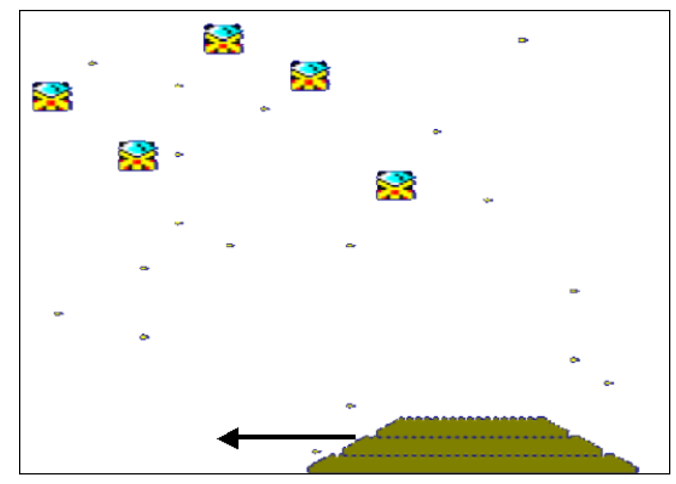


Fig. 27 un scroll horizontal de una montaña

aquí tienes el ejemplo que lo ilustra

```
1 MODE 0
10 MEMORY 26999: CALL &6B78
20 DEFINT a-z
30 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT
35' sprites de lateral derecho de la montana
40 FOR i=1 TO 3:|SETUPSP,i,9,&9102:|SETUPSP,i,6,-1:
|SETUPSP,i,0,&X1111:NEXT:'r
45 ' sprites para el lateral izquierdo de la montana
50 FOR i=7 TO 9:|SETUPSP,i,9,&9124:|SETUPSP,i,6,-1:
|SETUPSP,i,0,&X1111:NEXT:'l
55 x=80: inc=1
60 |LOCATESP,7,176,x:|LOCATESP,8,184,(x-4):|LOCATESP,9,192,(x-8)
70 |LOCATESP,1,176,x+20:|LOCATESP,2,184,(x+24):|LOCATESP,3,192,(x+28)
71' coloco 5 naves
75 FOR i=20 TO 24
:|SETUPSP,i,7,14:|SETUPSP,i,0,&X11101:|LOCATESP,i,RND*100,(i-
20)*10:NEXT :' relativo

79 ' logica principal
80 |PRINTSPALL,0,0
90 |AUTOALL
91 t=t+1: GOSUB 200
92 |STARS,0,20,1,0,-2
93 |MOVERALL,0,inc
94 tt=tt+1: GOSUB 300
100 GOTO 80

199' rutina que recoloca la montaña a la izquierda de la pantalla
200 IF t <140 THEN RETURN
210 x=100:|LOCATESP,7,176,x:|LOCATESP,8,184,(x-4):|LOCATESP,9,192,(x-
8)
220 |LOCATESP,1,176,x+20:|LOCATESP,2,184,(x+24):|LOCATESP,3,192,(x+28)
230 t=0:RETURN
300 IF tt=40 THEN inc=-inc: tt=0
310 RETURN
```

en el caso de un juego de scroll vertical, si queremos pintar un lago sobre un terreno marron, haremos lo mismo, unos sprites que van "manchando" el terreno y otros mas lejos que lo van "limpiando", aparentando que se trata de un lago enorme, de una sola pieza.

Solo debes tener una precaución, y es que las naves no sobrevuelen el lago o tu "truco" quedará al descubierto!

10.4 Caminando por el pueblo

Vamos a ver un último ejemplo que utiliza movimiento relativo para dar la sensación de scroll, usando sprites con dibujos de casas, un suelo moteado y un personaje situado en el centro que según la dirección que tome, hace que todo se mueva entorno a el. Es un

ejemplo muy básico pero te da una idea del potencial de estas funciones. Aquí lo que se mueve es todo el pueblo!

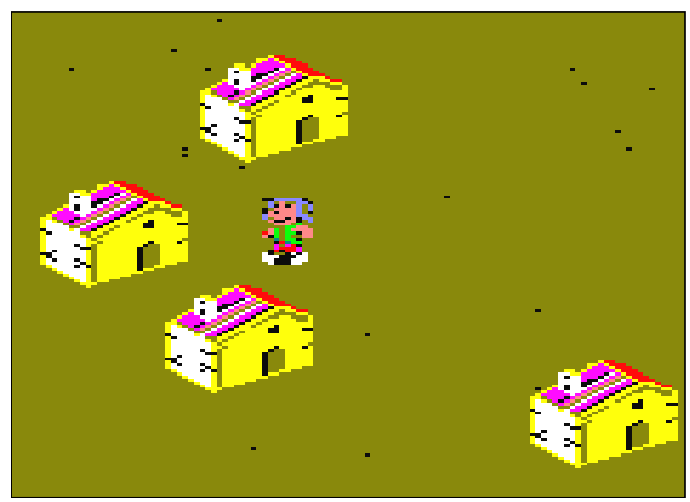


Fig. 28 El pueblo entero se mueve

```
10 MEMORY 26999
20 MODE 0: call &6b78
30 DEFINT a-z
240 INK 0,12
241 border 7
250 FOR i=0 TO 31
260 |SETUPSP,i,0,&X0
270 NEXT
280 FOR i=0 TO 3
290 |SETUPSP,i,0,&X10001
300 |SETUPSP,i,9,&A01c:rem casas
301 |LOCATESP,i,RND*150+50,rnd*60+10
310 NEXT
320 |SETUPSP,31,7,6: rem personaje
330 |LOCATESP,31,90,38
340 |SETUPSP,31,0,&X1111
400 xa=0:ya=0
410 IF INKEY(27)=0 THEN xa=-1:
420 IF INKEY(34)=0 THEN xa=+1:
430 IF INKEY(67)=0 THEN ya=+2
440 IF INKEY(69)=0 THEN ya=-2
450 |MOVERALL,ya,xa
460 |PRINTSPALL,1,0
470 |STARS,1,20,5,ya,xa
480 GOTO 400
```

11 Música

Las herramientas de las que voy a hablar en este apartado no las he hecho yo, pero están integradas en 8BP y son realmente buenas.

11.1 Editar musica con WyZ tracker

Esta herramienta es un secuenciador de musica para el chip de sonido AY3-8912. Las musicas que genera se pueden exportar y dan como resultado dos archivos

- Un archivo de instrumentos ".mus.asm"
- Un archivo de notas musicales ".mus"

Puedes componer canciones con esta herramienta y la única limitación que tendrás es que todas las canciones que integres en tu juego deberan compartir el mismo fichero de instrumentos

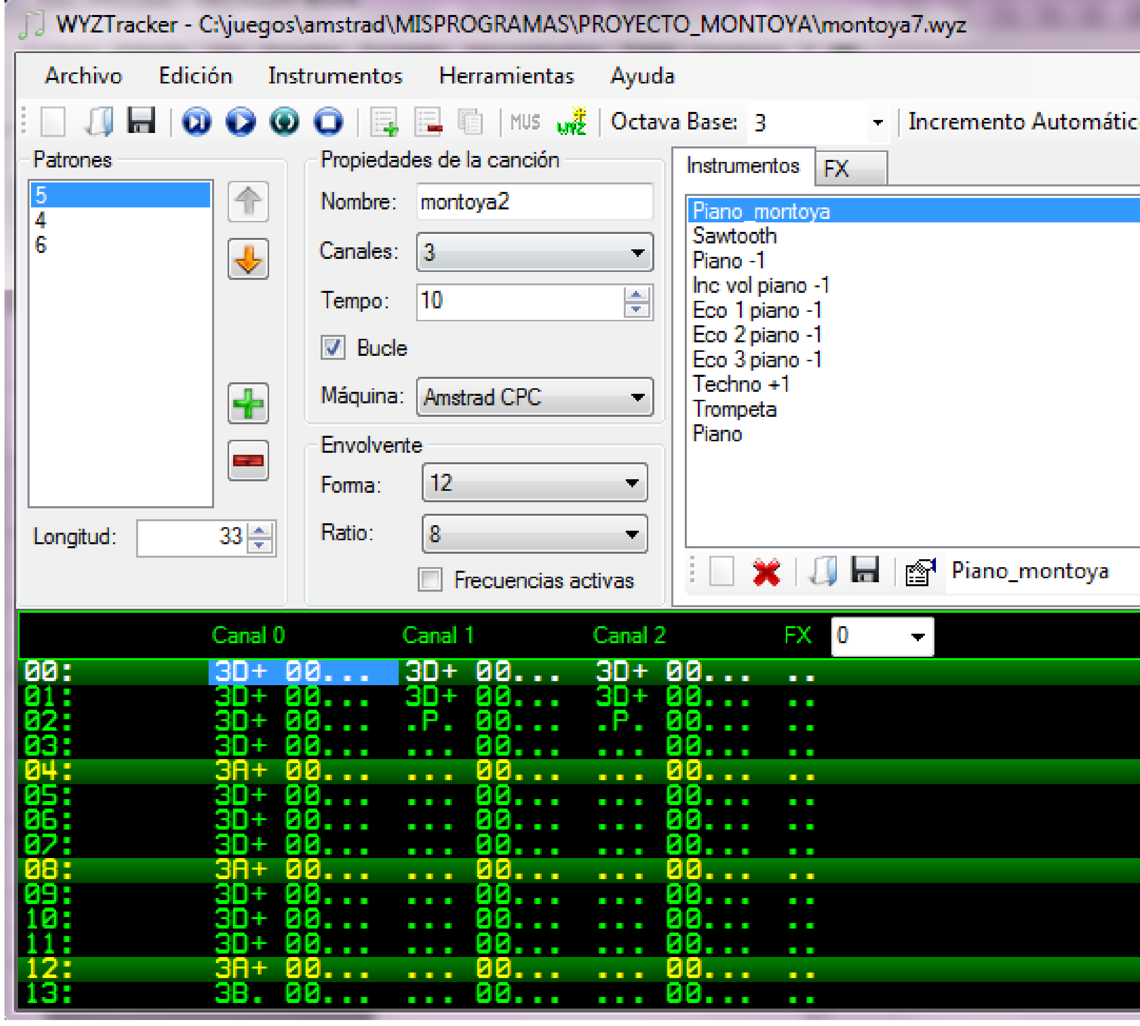


Fig. 29 WYZTracker

Este secuenciador de musica se complementa con el WyZplayer que está integrado en la librería 8BP.

Ls problemas que me he encontrado te los detallo aquí. Probablemente te ocurra lo mismo de modo que ya tienes las soluciones para que estas dificultades no te hagan sufrir

Al editar tu canción con WYZ tracker debes incluir una nota inicial en los 3 canales y en particular a mi me ha dado problemas si no incluia la nota D de la octava 3. Esto de incluir una primera nota que no forma parte de la musica debe servir para inicializar los 3 canales de sonido en el WYZplayer. Esta nota especial debe ser del instrumento con id=0
Cuando escuchas la canción en el WyZtracker, esa nota te estorbará pero no sonará en el player integrado en 8BP. Fijate en la captura de pantalla que he puesto, verás esa primera nota, tocada con el instrumento con id=0 que en este caso es "piano_montoya"

Este problema no tiene ningun efecto negativo si cumplimos esta regla. De lo contrario es posible que la canción no suene como debe o que haya canales que no suenen.

11.2 Ensamblar las canciones

Una vez que has compuesto tu canción y ya tienes los dos archivos, simplemente editas el fichero make_music.asm e incluyes tus ficheros de musica asi:

```
; si ensamblas esto independientemente
; deberia ser al menos donde acaba el codigo de 8bp y del player,
comprobando
; donde se ensambla la etiqueta _FIN_CODIGO.
; suponiendo que es menor de 32000 (en realidad es algo menos, puedes
ensamblar en 32000)
; tras ensamblarlo, salvalo con save "musica.bin",b,32000,1500
org 32000
;--------------------MUSICA------------------------------------
; tiene la limitacion de tan solo poder incluir un solo fichero de
; instrumentos para todas las canciones
; la limitacion se solventa simplemente metiendo todos los
; instrumentos en un solo fichero.

;archivo de instrumentos. OJO TIENE QUE SER SOLO UNO
read  "instrumentos.mus.asm"

; archivos de musica
; ojo la primera nota debe sonar en los 3 canales y ademas ya nunca se
repetira
; IMPORTANTE esta nota especial debe ser del instrumento con id=0
(edito usando WYZ tracker)
; si es de otro instrumento me da problemas.
SONG_0:
INCBIN      "micancion.mus" ;
SONG_0_END:

SONG_1:
INCBIN      "otra_cancion.mus" ;
SONG_1_END:
```

```
SONG_2:
INCBIN      "tercera_cancion.mus" ;
SONG_2_END:
SONG_3:
SONG_4:
SONG_5:
SONG_6:
SONG_7:
```

Por último re-ensamblas la librería 8BP para que el player de musica (que esta integrado en la librería) conozca los parámetros de instrumentos y el lugar donde han quedado ensambladas las canciones.

Para ello simplemente ensamblas el fichero make_all.asm, que tiene este aspecto

```
; Makefile para los videojuegos que usan 8bits de poder
; si alteras solo una parte solo tienes que ensamblar el make
correspondiente
; por ejemplo puedes ensamblar el make_graficos si cambias dibujos
;--------------------CODIGO ------------------------------
;incluye la libreria 8bp y el playerWYZ de musica
read "make_codigo.asm"

;--------------------MUSICA------------------------------------
; incluye las canciones.
read "make_musica.asm"

; ---------- GRAFICOS -------------------
; esta parte incluye imagenes y secuencias de animacion
; y la tabla de sprites inicializada con dichas imagenes y secuencias
read "make_graficos.asm"
```

y con esto ya tienes todo ensamblado. Ahora debes generar tu librería 8BP asi:
save “8BP.LIB”,b,27000,5000

y las musicas:

save “MUSIC.BIN”, b,32000,1500

12 Guía de referencia de la librería 8BP

12.1 Funciones de la librería

12.1.1 |ANIMA

Este comando cambia el fotograma de animación de un sprite, teniendo en cuenta su secuencia de animación asignada

Uso:
|ANIMA,<sprite number>

ejemplo

|ANIMA,3

El comando lo que hace es consultar la secuencia de animacion del sprite, y si es distinta de cero entonces se va a la tabla de secuencias de animacion (la primera secuencia valida es la 1 y la ultima es la 31). Escoge la imagen cuya posición es la siguiente al fotograma actual y actualiza el campo frame de la tabla de atributos de sprites.
Si en la secuenta el siguiente fotograma es cero entonces se cicla, es decir, se escoge el primer fotograma de la secuencia.

Además de cambiar el campo frame, se cambia el campo image y se le asigna la dirección de memoria del lugar donde se almacena el nuevo fotograma.

|ANIMA no imprime el sprite pero lo deja preparado para cuando se imprima, de modo que se imprima el siguiente fotograma de su secuencia

|ANIMA no verifica que el flag de animación esté activo en el byte de estado del sprite. De hecho nuestro personaje normalmente solo lo vamos a querer animar cuando se mueva y no siempre que se imprima.

12.1.2 |ANIMALL

Este comando anima todos los sprites que tengan el flag de animación activado en el byte de estado. Este comando no tiene parámetros

Uso
|ANIMALL

Es recomendable su uso si vas a animar muchos sprites ya que es mucho mas rapido que invocar varias veces al comando |ANIMA

Como normalmente se va a desear invocar a ANIMALL en cada ciclo de juego, antes de imprimir los sprites, hay una forma de invocar mas eficiente y consiste en poner a “1” el primer parámetro de la funcion PRINTSPALL, es decir

|PRINTSPALL,1,0

Esta función invoca internamente a ANIMALL antes de imprimir los sprites, ahorrando 1.17ms respecto de lo que se tardaría en invocar separadamente |ANIMALL y |PRINTSPALL

12.1.3 |AUTO

Este comando mueve un sprite (cambia sus coordenadas) de acuerdo a sus atributos de velocidad Vy,Vx

Uso:
|AUTO,<sprite number>

Ejemplo:
|AUTO,5

Lo que hace este comando es actualizar las coordenadas en la tabla de sprites, sumando la velocidad a la coordenada actual

Las coordenadas nuevas son
X nueva = coordenada X actual + Vx
Y nueva = coordenada Y actual + Vy

No es necesario que el sprite tenga el flag de movimiento automatico activo en el campo status

12.1.4 |AUTOALL

Este comando mueve todos los sprites que tengan el flag de movimiento automatico activo, de acuerdo a sus atributos de velocidad Vy, Vx. Este comando no tiene parámetros

Uso:
|AUTOALL

12.1.5 |COLAY

Detecta la colisión de un sprite con el mapa de pantalla (el layout). Tiene en cuenta el tamaño de dicho sprite para saber si colisiona.

Uso:
|COLAY, <num_sprite>,<@colision%>

Ejemplo:
|COLAY, 0,@colision%

La variable que uses para colisión puede llamarse como quieras.

Esta rutina modifica la variable colision (la cual debe ser entera y por eso el “%”) poniéndola a 1 si hay colision del sprite indicado con el layout. Si no hay colision el resultado es 0.

xanterior=x
x=x+1
|LOCATESP,0,y,x: ‘ posicionamos el sprite en nueva posicion
|COLAY,0,@colision%: ‘chequeo de la colision

Ahora comprobamos la colision y si hay colision lo dejamos en su ubicación anterior

if colision%=1 then x=xanterior: LOCATESP,0,y,x

Si nuestro personaje puede moverse en diagonal, a menudo querremos que al pulsar derecha+arriba nuestro sprite avance en una dirección aunque en la otra haya un muro y quede bloqueado. Esto da una sensación de mayor fluidez al movimiento aunque complica la lógica. Aquí se muestra como hacerlo, a partir de la línea 1721. Primero se coloca al sprite con LOCATESP y luego en función de las colisiones que se detectan con COLAY se recoloca nuevamente el sprite con LOCATESP

En este ejemplo las variables tienen el siguiente significado
Xa: coordenada x anterior
ya: coordenada y anterior
Xn: coordenada x nueva
yn: coordenada y nueva

```
1500' rutina de movimiento personaje-------------------------
1510 if inkey(27)<>0 goto 1520

1511 if inkey(67)=0 then IF dir<>2 THEN |SETUPSP,0,7,2:dir=2:goto 1533
ELSE |ANIMA,0:xn=xa+1:yn=ya-2:goto 1533

1512 if inkey(69)=0 THEN IF dir<>8 THEN |SETUPSP,0,7,8:dir=8:goto 1533
ELSE |ANIMA,0:xn=xa+1:yn=ya+2:goto 1533

1513 IF dir<>1 THEN |SETUPSP,0,7,1:dir=1:goto 1533 ELSE
|ANIMA,0:xn=xa+1:goto 1533

1520 if inkey(34)<>0 goto 1530
```

```
1521 if INKEY(67)=0 THEN IF dir<>4 THEN |SETUPSP,0,7,4:dir=4:goto 1533
ELSE |ANIMA,0:xn=xa-1:yn=ya-2:goto 1533

1522 if INKEY(69)=0 THEN IF dir<>6 THEN |SETUPSP,0,7,6:dir=6:goto 1533
ELSE |ANIMA,0:xn=xa-1:yn=ya+2:goto 1533

1523 IF dir<>5 THEN |SETUPSP,0,7,5:dir=5:goto 1533 ELSE
|ANIMA,0:xn=xa-1:goto 1533
```

```
1530 IF INKEY(67)=0 THEN IF dir<>3 THEN |SETUPSP,0,7,3:dir=3:goto 1533
ELSE |ANIMA,0:yn=ya-4:goto 1533

1531 IF INKEY(69)=0 THEN IF dir<>7 THEN |SETUPSP,0,7,7:dir=7:goto 1533
ELSE |ANIMA,0:yn=ya+4:goto 1533

1532 return

1533 |LOCATESP,0,yn,xn:ynn=yn:|COLAY,0,@cl%:IF cl%=0 then 1450

1534 yn=ya:|LOCATESP,0,yn,xn:|COLAY,0,@cl%:IF cl%=0 then 1450

1535 xn=xa: yn=ynn:|LOCATESP,0,yn,xn:|COLAY,0,@cl%:IF cl%=1 THEN
yn=ya:|LOCATESP,0,yn,xn

1536 ya=yn:xa=xn

1537 return
```

12.1.6 |COLSP

este comando permite detectar la colision de un sprite con el resto de sprites que tengan el flag de colisión activo

uso
|COLSP,<sprite number>, @colsp%

ejemplo
col%=0
|COLSP,0,@col%

la funcion retorna en la variable que le pasemos como parámetro, el número del sprite con el que colisiona, o si no hay colisión retorna un 32 pues el sprite 32 no existe (solo existen del 0 al 31)

Al igual que la impresión de sprites con PRINTSPALL, la funcion COLSP chequea los sprites comenzando en el 31 y acabando en el cero. Si tienen flag de colisión de sprites activo (bit 2 del byte de status) entonces se comprueba la colision. Di dos sprites colisionan a la vez con nuestro sprite, se retorna el número de sprite mayor pues es el que se comprueba antes.

12.1.7 |LAYOUT

uso:
|LAYOUT, <y>,<x>, <@cadena$>

Ejemplo:
cadena$=”XYZZZZ ZZ”
|LAYOUT, 0,1, @cadena$

ojo, usar |LAYOUT, 0,1, ”XYZZZZ ZZ” sería incorrecto en un CPC464 aunque funciona en un CPC6128. Además, en cpc6128 puedes obviar el uso de la “@” pero en CPC464 es obligatorio.

Esta rutina imprime una fila de sprites para construir el layout o "laberinto" de cada pantalla. Además de dibujar el laberinto, o cualquier gráfico en pantalla construido con pequeños sprites de 8x8, también podrás detectar las colisiones de un sprite con el layout, usando el comando |COLAY

Los sprites a imprimir se definen con un string, cuyos caracteres (32 posibles) representan a uno de los sprites siguiendo esta sencilla regla, donde la única excepción es el espacio en blanco que representa la ausencia de sprite.

Caracter	Sprite id	Codigo ASCII
“;”	0	59
“<”	1	60
“=”	2	61
“>”	3	62
“?”	4	63
“@”	5	64
“A”	6	65
“B”	7	66
“C”	8	67
“D”	9	68
“E”	10	69
“F”	11	70
“G”	12	71
“H”	13	72
“I”	14	73
“J”	15	74
“K”	16	75
“L”	17	76
“M”	18	77
“N”	19	78
“O”	20	79
“P”	21	80
“Q”	22	81
“R”	23	82
“S”	24	83
“T”	25	84
“U”	26	85
“V”	27	86
“W”	28	87
“X”	29	88
“Y”	30	89
“Z”	31	90
“ “	NINGUNO	128

Tabla 8 correspondencia entre caracteres y Sprites para el comando |LAYOUT

IMPORTANTE: Tras imprimir el layout puedes cambiar los sprites para que sean personajes, por lo que seguirás disponiendo de los 32 sprites

Las coordenadas y,x se pasan en formato caracteres. La librería mantiene internamente un mapa de 20x25 caracteres, por lo que las coordenadas toman los siguientes valores:

y toma valores [0,24]

x toma valores [0,19]

Los sprites a imprimir deben ser de 8x8 píxeles. son "ladrillos" ("bricks" en ingles).A este tipo de concepto también se le suele llamar “tiles” (azulejos)

Si usas otros tamaños de sprite, esta función no funcionará bien. Realmente imprimirá los sprites pero si un sprite es grande tendras que colocar espacios en blanco para dejarle espacio.

La librería mantiene un mapa interno del layout y esta función actualiza los datos del mapa interno del layout de modo que será posible detectar colisiones. Dicho mapa es un array de 20x25 caracteres, donde cada carácter se corresponde con un sprite

El @string es una variable de tipo cadena. no puedes pasar directamente la cadena, aunque en el 6128 el paso de parámetros lo permite pero sería incompatible con 464

Precauciones:

la función no valida la cadena que le pasas. Si contiene minúsculas u otro carácter diferente de los permitidos puede provocar efectos indeseados, tales como el reinicio o cuelgue del ordenador. Tampoco puede ser una cadena vacia!

Los limites establecidos con SETLIMITS deben permitir que se imprima donde desees. Si posteriormente quieres hacer cliping en una zona mas reducida puedes invocar de nuevo a SETLIMITS cuando todo el layout este impreso

Ejemplo

```
2070 |SETLIMITS,0,80,0,200
2090 c$(1)= "PPPPPPPPPPPPPPPPP  P"
2100 c$(2)= "PU                 P"
2110 c$(3)= "P                  P"
2120 c$(4)= "P                  P"
2130 c$(5)= "P  TPPPPU  TPPPPPPPP"
2140 c$(6)= "P                 TP"
2150 c$(7)= "P                  P"
2160 c$(8)= "P                  P"
2170 c$(9)= "P    YYYYYYYYYY    P"
2190 c$(10)="P    TPPPPPPPPU    P"
2195 c$(11)="P                  P"
2200 c$(12)="P                  P"
2210 c$(13)="P                  P"
2220 c$(14)="YYYYYYYYYP   PYYYYYY"
2230 c$(15)="RRRRRRRRRR   RRRRRRR"
2240 c$(16)="PPPPPPPPPP   PPPPPPP"
2250 c$(17)="PU      TP   PU   TP"
2260 c$(18)="P        T   U     P"
2270 c$(19)="P                  P"
2271 c$(20)="P                  P"
2272 c$(21)="P              W   P"
2273 c$(22)="PP             W  PP"
2274 c$(23)="PPPPPPPPPPPPPPPPPPPP"
```

En este ejemplo se usan varios ladrillos que previamente se han creado con la herramienta “spedit”

; sprite 20 --> O arbusto
; sprite 21 --> P roca
; sprite 22 --> Q nube
; sprite 23 --> R agua
; sprite 24 --> S ventana
; sprite 25 --> T arco de puente derecho
; sprite 26 --> U arco de puente izq
; sprite 27 --> V bandera
; sprite 28 --> W planta
; sprite 29 --> X pico de torre
; sprite 30 --> Y cesped
; sprite 31 --> Z ladrillo

```
2280 for i=0 to 23
2281|LAYOUT,i,0,@c$(i)
2282 next
```

12.1.8 |LOCATESP

Este comando cambia las coordenadas de un sprite en la tabla de atributos de sprites

Uso
|LOCATESP,<sprite number>, <y>,<x>

Ejemplo
|LOCATESP,0,10,20

Una alternativa a este comando, si solo deseamos cambiar una coordenada es usar el comando POKE de BASIC, insertando en la dirección de memoria ocupada por la coordenada X o Y, el valor que queramos. Si deseamos introducir una coordenada negativa es necesaria la funcion |POKE, ya que con el POKE de BASIC sería ilegal

El comando |LOCATE no imprime el sprite, sólo lo posiciona para cuando sea impreso.

12.1.9 |MOVER

Este comando mueve un sprite de forma relativa, es decir, sumando a sus coordenadas unas cantidades relativas

Uso:
|MOVER,<sprite number>, <dy>,<dx>

Ejemplo:
|MOVER,0,1,-1

El ejemplo mueve el sprite 0 hacia abajo y hacia la derecha a la vez. No es necesario que el sprite tenga el flag de movimiento relativo activado

12.1.10 |MOVERALL

este comando mueve de forma relativa todos los sprites que tengan el flag de movimiento relativo activado

Uso
|MOVERLALL,<dy>,<dx>

Ejemplo
|MOVERALL,2,1

El ejemplo mueve todos los sprites con flag de movimiento relativo hacia abajo (2 lineas) y 1 byte hacia la derecha.

12.1.11 |MUSIC

Este comando permite que una melodía comience a sonar

Uso:
|MUSIC,<numero_melodía>,<velocidad>

el numero de la melodía estará comprendido entre 0 y 7
la velocidad “normal” es 5. si usamos un numero superior se reproducirá más lentamente y si el numero es inferior se reproducirá más deprisa

Ejemplo:
|MUSIC,0,5

internamente el comando lo que hace es instalar una interrupción que se dispara 300 veces por segundo. Si ponemos velocidad 5, una de cada 5 veces que se dispara, se ejecuta la función de reproducción musical

12.1.12 |MUSICOFF

Este comando paraliza la reproducción de cualquier melodía. No tiene parámetros

Uso:
|MUSICOFF

Internamente lo que hace es desinstalar la interrupción

12.1.13 |PEEK

Este comando lee el valor de un dato de 16 bit de una direccion de memoria dada. Esta pensado para consultar las coordenadas de sprites que se mueven con movimiento automático o relativo

Uso
|PEEK,<dirección>, @dato%

Ejemplo
dato%=0
|PEEK, 27001, @dato%

si las coordenadas son solo positivas y menores de 255 puedes usar el comando PEEK de BASIC, ya que es algo mas rápido.

12.1.14 |POKE

Este comando introduce un dato de 16 bit (positivo o negativo) en una direccion de memoria. Esta pensada para modificar coordenadas de sprites, ya que el comando POKE no puede manejar coordenadas negativas o mayores de 255 ya que POKE funciona con bytes mientras que |POKE es un comando que funciona con 16bit

Uso:
|POKE,<dirección>,<valor>

Ejemplo:
|POKE,27003,23

Este ejemplo pone el valor 23 en la coordenada x del sprite 0.
Es una función muy rápida aunque si vas a manejar solo coordenadas positivas es mejor usar POKE pues es más rápida aun

12.1.15 |PRINTSP

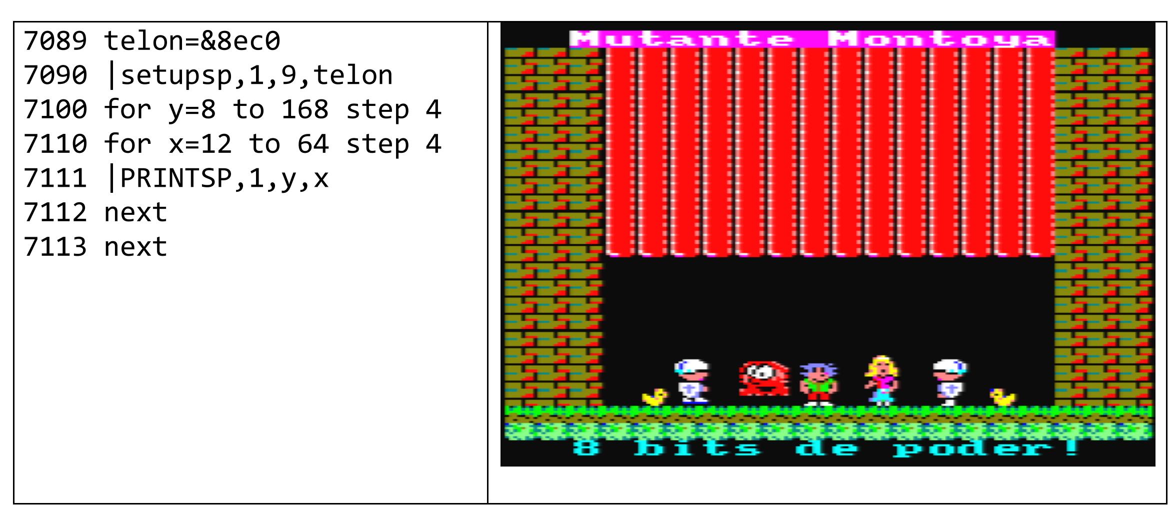
Uso:
|PRINTSP, <sprite id >, <y >,<x>

Ejemplo
imprime el sprite 23 en las coordenadas y=100, x=40

|PRINTSP, 23,100,40

Esta rutina imprime un sprite en la pantalla, pero no por ello actualiza las coordenadas del sprite en la tabla de sprites.
Las coordenadas consideradas son
Numero de líneas en vertical [-32768..32768]. las correspondientes al interior de la pantalla son [0..199]
Numero de bytes en horizontal [-32768..32768]. las correspondientes al interior de la pantalla son [0..79]

Normalmente en la lógica de un vieojuego harás uso de |PRINTSPALL, ya que es mas rápido imprimirlos todos de golpe. Sin embargo en otros momentos del juego puede interesarte imprimir sprites por separado. En este ejemplo se muestra la bajada de un "telón", usando un solo sprite que se repite horizontalmente y al ir bajando va "tiñendo" de rojo la pantalla, dando la sensación de un telón que baja



```
7089 telon=&8ec0
7090 |setupsp,1,9,telon
7100 for y=8 to 168 step 4
7110 for x=12 to 64 step 4
7111 |PRINTSP,1,y,x
7112 next
7113 next
```


Fig. 30 Un ejemplo de uso de PRINTSP

12.1.16 |PRINTSPALL

Uso:
|PRINTSPALL, <flag anima>, <flag sync>

Ejemplo:
imprime todos los sprites animándolos primero y sin sincronizar con barrido
|PRINTSPALL, 1, 0

Esta rutina imprime de una sola vez todos los sprites que tengan el bit0 de estado activo. Si se pone a 1 el flag de animación, entonces antes de imprimir los sprites se cambia el fotograma en su secuencia de animación, siempre que los sprites tengan el bit 3 de estado activo

El <flag sync> es un flag de sincronización con el barrido de pantalla. Puede ser 1 o 0 . La sincronización solo tiene sentido si compilas el programa con un compilador como “Fabacom”. La lógica en BASIC se ejecuta lentamente y sincronizar con el barrido produce pequeñas esperas adicionales en cada ciclo del juego de modo que no es conveniente.
Como regla general, solo es conveniente si tu juego es capaz de generar 50fps por segundo, o lo que es lo mismo, un ciclo completo de juego cada 20 milisegundos. Si compilas el juego con un compilador como “fabacom”, entonces es recomendable que sincronices con el barrido de pantalla porque casi seguro que vas a alcanzar esos 50fps y si los superas, tu juego producirá mas fotogramas de los que puede mostrar la pantalla y entonces algunos no se podrán mostrar y el movimiento no será suave.

Cuantos más sprites tengas en pantalla imprimiéndose, más tardará el comando, aunque es muy rápido. Hay muchos sprites que pueden aparecer en pantalla pero no es necesario imprimir (pueden tener el bit 0 de estado desactivado) como pueden ser frutas, monedas, elementos de bonus en general y/o personajes que no se mueven y no tienen animación. Aunque no se impriman pueden tener el bit de colisión activo y asi afectar en la rutina |COLSP

12.1.17 |SETLIMITS

Este comando establece los límites del area donde se van a poder imprimir sprites o estrellas.

Uso:
|SETLIMITS,<xmin>,<xmax>,<ymin>,<ymax>

Ejemplo que establece toda la pantalla como area permitida
|SETLIMITS,0,80,0,200

Fuera de estos límites se realiza clipping de los sprites, de modo que si un sprite se encuentra parcialmente fuera del area permitida, las funciones |PRINTSP y |PRINTSPALL imprimirán solo la parte que se encuentra dentro del área permitida.

12.1.18 |SETUPSP

Este comando carga datos de un sprite en la SPRITES_TABLE
Uso:
|SETUPSP, <id_sprite>,<param_number>,<valor>

Ejemplo
|SETUPSP,3,7,2

Permite por ejemplo asignar una nueva secuencia de animación cuando el sprite cambia de direccion, o simplemente cambiar su registro de flags de status
;Con esta funcion podemos cambiar cualquier parámetro de un sprite, menos X, Y (que se hace con LOCATE_SPRITE)
Solo podremos cambiar un parametro a la vez. El parametro que vamos a cambiar se especifica con param_number. El param_number es en realidad la posicion relativa del parametro en la SPRITES_TABLE

param_number=0 --> cambia el status (ocupa 1 byte)
param_number=5 --> cambia Vy (ocupa 1byte, valor en lineas verticales)
param_number=6 --> cambia Vx (ocupa 1byte, valor en bytes horizontales)
param_number=7 --> cambia secuencia (ocupa 1byte, toma valores 0..31)
param_number=8 --> cambia frame_id (ocupa 1byte, toma valores 0..7)
param_number=9 --> cambia dir imagen (ocupa 2bytes)

ejemplo
en este ejemplo le hemos dado al sprite 31 la imagen de una nave que está ensamblada en la dirección &a2f8

nave = &a2f8
|SETUPSP,31,9,nave

12.1.19 |SETUPSQ

Este comando crea una secuencia de animación

Uso:
|SETUPSQ,<numero de secuencia>, <dirección1>, <direccionN>, 0,…,0
Se deben rellenar 8 direcciones o completar hasta 8 direcciones con ceros
El numero de secuencia debe estar entre 1 y 31

Ejemplo que crea la secuencia 1 con las 4 direcciones donde hay ensambladas las imágenes (fotogramas) de un personaje animado

SETUPSQ,1, &926c,&92FE,&9390 ,&02fe ,0,0,0,0

12.1.20 |STARS

Mueve un banco de hasta 20 estrellas en la pantalla (dentro de los límites establecidos por |SETLIMITS), sin pintar sobre otros sprites que ya existiesen impresos.

|STARS,<banco>,<num estrellas>,<color>,<dy>,<dx>

Ejemplo

|STARS,0,15,3,1,0

El ejemplo desplaza 15 estrellas de color 3 (rojo) un píxel verticalmente (ya que dy=1 y dx=0). Invocado repetidas veces da sensación de fondo de estrellas que se desplaza. Cuando una estrella se sale del limite de la pantalla o el establecido por |SETLIMITS, reaparece por el lateral opuesto, de modo que hay sensación de continuidad en el fluir de las estrellas.

Puedes manejar 2 bancos de estrellas diferentes, que están identificados con el “0” y el “1”. Cada estrella ocupa 2 bytes, uno para la coordenada y [0..200] y otro para la coordenada x [0..79].

Las coordenadas iniciales de las estrellas deben ser inicializadas por el programador, teniendo en cuenta que banco de estrellas va a usar.
Banco 0: desde la dirección 42540 hasta 42579 (son 40 bytes)
Banco 1: desde la dirección 42580 hasta 42619 (son 40 bytes)

Ejemplo de inicialización y uso en un doble scroll de estrellas para dar sensación de profundidad

```
30 ' inicializacion banco 0
40 FOR dir=42540 TO 42578 STEP 2: POKE dir,RND*200:POKE
dir+1,RND*80:NEXT
30 ' inicializacion banco 1
```

```
40 FOR dir=42580 TO 42618 STEP 2: POKE dir,RND*200:POKE
dir+1,RND*80:NEXT
50 |STARS,0,20,1,2,0
60 |STARS,1,20,2,1,0
70 GOTO 50
```

Los usos de este comando pueden ser muy diversos.

- Usando los dos bancos de estrellas a la vez con diferente velocidad y color puedes dar sensación de profundidad
- Si la dirección de las estrellas es diagonal puedes hacer un “efecto de lluvia”
- Si el color es negro y el fondo es marrón o naranja puedes dar sensación de avance sobre un territorio arenoso
- Si el movimiento es de balanceo y el color de las estrellas es blanco puedes dar sensación de nieve. El movimiento de balanceo lo puedes lograr con un zigzag en X manteniendo la velocidad en Y, o incluso usando funciones trigonométricas como el coseno. Obviamente si usas el coseno en la lógica de un juego va a ser muy lento pero puedes almacenar el valor del coseno precalculado en un array.

Ejemplo de efecto nieve:

```
1 MODE 0
30 ' inicializacion banco 1
40 FOR dir=42580 TO 42619 STEP 2: POKE dir,RND*200:POKE
dir+1,RND*80:NEXT
50 |STARS,0,20,4,2,dx1
60 |STARS,1,20,4,1,dx2
61 dx1=1*COS(i)
62 dx2=SIN(i)
69 i=i+1: IF i=359 THEN i=0
70 GOTO 50
```

13 Ensamblado de la librería, gráficos y música

Tanto si quieres hacer cambios en la librería como si añades música y gráficos deberás reensamblarla, Esto es debido a que por ejemplo, el player de música esta integrado en la librería y necesita conocer donde comienza (dirección de memoria) cada canción, por lo que es necesario reensamblar y guardar la versión de la librería especifica para tu juego, así como el fichero de gráficos ensamblado y el fichero de música ensamblado.

Como expliqué en el apartado de los "pasos" que debes dar, ésta será una versión de la librería especifica para tu juego. Por ejemplo el comando |MUSIC,3,5 hará sonar la melodía número 3 que tu mismo has compuesto. La melodía numero 3 puede ser completamente diferente en otro juego. Lo mismo ocurre con los datos del fichero de instrumentos. Hay ciertas dependencias entre el codigo del player de música y las direcciones donde se ensamblan los datos de instrumentos y las melodías.
Es muy sencillo pero hay que comprender la estructura de la librería para hacerlo

Lo primero que debes tener claro es que tu juego se compone de 3 ficheros binarios:

- Librería 8BP (es un fichero binario), incluyendo la tabla de atributos de sprites
- Fichero binario de musica, con las melodías de tu juego
- Fichero binario de imágenes de sprites, incluyendo la tabla de secuencias de animación

Y dos ficheros BASIC:

- Cargador (carga la librería, musica y sprites y por último tu juego)
- Programa BASIC (tu juego)

En este apartado vamos a centrarnos en como generar los 3 ficheros binarios que necesitas. Para generar los 3 ficheros primero debes ensamblarlo todo (ahora te diré como) y después que tengas todo ensamblado en memoria ejecutas estos comandos para generar los ficheros. Como ves estos comandos simplemente toman fragmentos de la memoria y la salvan en ficheros independientes.

```
SAVE “8BP.LIB”,b,27000,5000
SAVE “MUSIC.BIN”,b,32000,1500
SAVE “SPRITES.BIN”,b,33500,8500
```

Ahora queda comprender como ensamblar todo en memoria (librería, musica y graficos). Para ello debes comprender la estructura de los ficheros .asm que debes manejar y sus dependencias. Un solo fichero .BIN en realidad va a requerir de más de un fichero .asm para generarlo.

El siguiente diagrama presenta todos los ficheros .asm de un juego que use 8BP asi como las dependencias entre ellos.

En gris aparecen aquellos ficheros que tienes que editar, como son:
las canciones y fichero de instrumentos, que generas con el WYZtracker
el fichero make_musica donde indicas que ficheros .mus hay que ensamblar
el fichero de imágenes que creas con el SPEDIT
la tabla de sprites donde asignas imágenes a los sprites (aunque no es estrictamente necesario pues tienes el comando |SETUPSP)
la tabla de secuencias, donde defines que imágenes conforman una secuencia, (aunque no es estrictamente necesario pues tienes el comando |SETUPSQ)

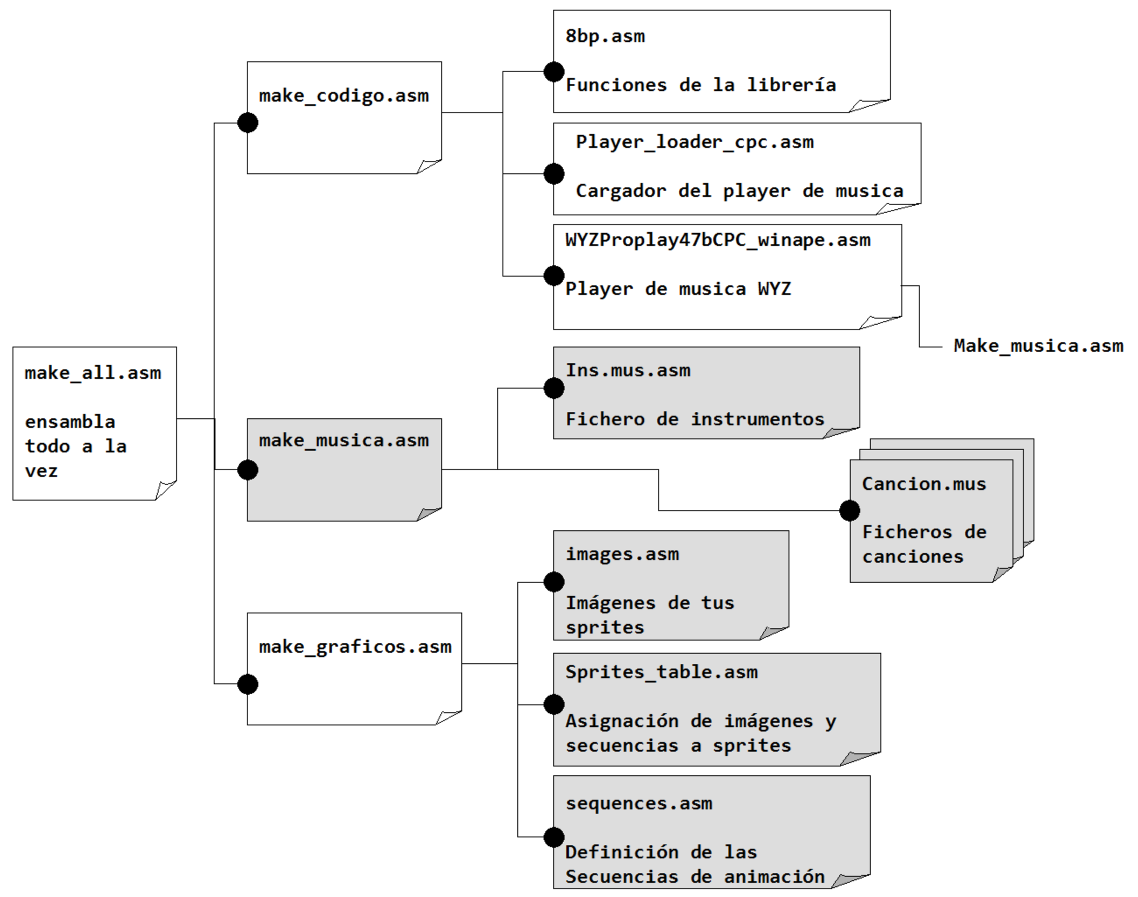


Fig. 31 Ficheros para ensamblar

Puedes ensamblarlo todo con make_all y después usar el comando SAVE para salvar las imágenes, musica y librería 8BP en diferentes ficheros binarios, tal como hemos visto. Para ello simplemente abres el fichero make_all dentro del editor de winape y pulsas "assemble"

Si sólo cambias los gráficos puedes ensamblarlos por separado, seleccionando el fichero "make_graficos.asm" y pulsando assemble

Si cambias las musicas debes re-ensamblar el código de la librería pues hay una dependencia entre el codigo y las canciones, debido a que el código necesita conocer donde comienza cada canción. Por ello si cambias o añades canciones debes ensamblar con make_all.asm y volver a salvar tus ficheros "8BP.LIB" y "MUSIC.BIN"

14 Posibles mejoras futuras a la librería

La librería 8BP es mejorable, añadiendo nuevas funciones que podrían abrir nuevas posibilidades para el programador. Aquí se muestran algunas sugerencias para hacerlo

14.1 Memoria para ubicar nuevas funciones

Actualmente hay espacio para unos 450 bytes de nuevas funciones sin reducir el espacio disponible para el programador (27KB). Este espacio se encuentra entre 31550 y 32000

En el caso de requerir aun más espacio para nuevas funciones, una opción sería reducir el espacio para la música a 1KB lo cual dejaría espacio para 500 bytes de nuevas funciones, todo ello sin reducir el espacio disponible para el programador (27KB). Esto limita las posibilidades musicales pero 1KB es espacio suficiente para albergar un par de grandes músicas o 4 cortas.

Otra posibilidad es reducir los 8KB destinados a gráficos. Y la última alternativa es la de reducir la memoria al programador de BASIC, pero esa debería ser la última opción.

14.2 Ahorrar memoria de gráficos

Una posible operación
|FLIP, secuencia, dirección
Podría dar la vuelta a todos los frames de una secuencia de animación, ahorrando mucha memoria de gráficos. Sería una instrucción fácil de programar y muy rentable en el ahorro. Al programador le obligaría a invocarla cuando un sprite cambia de dirección o bien, para evitar cualquier ralentización por ese motivo, se podría hacer FLIP de todos los tipos de enemigos que fuesen a aparecer en una pantalla justo antes de entrar en ella, reutilizando la memoria en otras pantallas con otros enemigos diferentes.

14.3 Impresión a resolución de píxel

Actualmente usa resolución de bytes y coordenadas de byte, que son 2 pixels de mode 0

14.4 Impresión con sobreescritura

La rutina de impresión actual no soporta sobreescritura por lo que los fondos de los juegos deben ser lisos. Una nueva funcion o un parámetro adicional en las funciones actuales PRINTSP y PRINTSPALL sería una buena mejora

Obviamente una impresión con sobreescritura es mas lenta pero en muchos juegos daría un aspecto mas profesional

14.5 Layout de mode 1

El layout actual funciona como un buffer de caracteres de 20x25 = 500Bytes
Se puede usar en juegos en mode 1 sin problemas pero habrá cosas que no podamos hacer, como definir una pieza que ocupe 3 caracteres de ancho de mode 1, ya que los

caracteres de mode 0 ocupan el doble de ancho de los de mode 1. No es un problema, pero sí es una limitación.

Un layout de mode 1 ocuparía 1KB pues 40x25=1000 . Puesto que el layout de mode 0 y el de mode 1 no se usarían simultáneamente, podrían solaparse en memoria y teniendo en cuenta que el de mode 0 esta en 42000 hasta 42500, simplemente el de mode 1 lo situaríamos entre 41500 y 42500, “robando” 500bytes a la memoria de sprites de 8KB, situada entre 34000 y 42000

Los cambios para soportar esta mejora son mínimos, afectando solo a dos funciones |LAYOUT y |COLAY que deberían ser conscientes del modo de pantalla, mediante una variable que actuase a modo de flag (layout0/layout1) y que manipulásemos desde BASIC con POKE

14.6 Funciones de scroll

Existen pocos juegos de amstrad CPC con un scroll suave de calidad, programado usando las capacidades del chip controlador de video M6845

El hecho de que no existan muchos juegos así responde al hecho de que en los años 80 los programadores de videojuegos no tenían mucha información y además en muchos casos eran aficionados

Entre los pocos juegos que tienen scroll suave destacan 2 de firebird

- Misión genocide (de firebird, 1987, por Paul Shirley, un excelente programador que además inventó una técnica de sobrescritura ultrarrápida sin uso de máscaras)
- Warhawk (de firebird, 1987)



Fig. 32 Juegos de firebird con scroll rápido y suave

Incorporar un scroll de esta calidad a la libreria 8BP sería muy valioso.

14.7 Migrar la librería 8BP a otros microordenadores

Esta librería seía fácilmente portable a otros microordenadores basados en el Z80, como el Sinclair ZX Spectrum. En el caso del ZX spectrum habría que rescribir las rutinas que pintan en pantalla pues la memoria de video se maneja de modo diferente.

La migración de la librería a un Commodore 64 también sería factible, aunque no se podría reutilizar el código ensamblador, ya que está basado en otro microprocesador. Además, en el caso del commodore 64, la migración de la librería 8BP debería aprovechar las características propias de la máquina como sus 8 sprites hardware, de modo que lo que debería incorporar internamente la librería 8BP sería un sprite multiplexer , ofreciendo 32 sprites pero internamente usando los 8 sprites hardware.



Fig. 33 Sinclair ZX y Commodore 64, dos clásicos

15 Apéndice I: código del Mutante Montoya

En este apéndice se han incluido todos los ficheros del juego salvo el que contiene los bytes de las imágenes, por ser muy grande (images_montoya.asm)



Fig. 34 Pantalla inicial del "Mutante montoya"

15.1 Ficheros BASIC

15.1.1 Cargador (loader.bas)

```
10 MODE 1
20 DEFINT a-z
21 CALL &BC02:REM paleta default
22 INK 0,0
30 a=3:b=1:c=3: INK 0,0:INK 1,0:INK 2,0:INK 3,0:BORDER 0
40 SYMBOL AFTER 0
50 SYMBOL 111,&X001111,&X0111111,255,255,255,255,&X0111111,&X001111
60 SYMBOL
111,0,&X001111,&X0111111,&X0111111,&X0111111,&X0111111,&X001111,0
70 PEN a
80 PRINT"oooooooooooooooooooooooooooooooooooooooo";:PEN b
90 PRINT"o o o o o o o o o o o o o o o o o o o o ";:PEN a
100 PRINT"oooooooooooooooooooooooooooooooooooooooo";:PEN c
110 PRINT" o   o o  o ooooo  oo  o  o ooooo oooo  ";
120 PRINT" oo oo o  o   o   o  o oo o   o   o     ";
130 PRINT" o o o o  o   o   oooo o oo   o   oo    ";
140 PRINT" o   o o  o   o   o  o o  o   o   o     ";
```

```
150 PRINT" o   o  oo    o   o  o o  o   o   oooo  ";:PEN a
160 PRINT"oooooooooooooooooooooooooooooooooooooooo";:PEN b
170 PRINT"o o o o o o o o o o o o o o o o o o o o ";:PEN a
180 PRINT"oooooooooooooooooooooooooooooooooooooooo";:PEN c
190 PRINT" o   o  oo  o  o ooooo  oo  o   o  oo   ";:
200 PRINT" oo oo o  o oo o   o   o  o  o o  o  o  ";
210 PRINT" o o o o  o o oo   o   o  o   o   oooo  ";
220 PRINT" o   o o  o o  o   o   o  o   o   o  o  ";
230 PRINT" o   o  oo  o  o   o    oo    o   o  o  ";:PEN a
240 PRINT"oooooooooooooooooooooooooooooooooooooooo";:PEN b
250 PRINT"o o o o o o o o o o o o o o o o o o o o ";:PEN a
260 PRINT"oooooooooooooooooooooooooooooooooooooooo";:PEN c
270 PRINT"":SYMBOL AFTER 0
280 PRINT"                is loading ...."
290 PRINT""
300 PRINT"         Jose Javier Garcia Aranda 2016 ";:PEN 2
310 PRINT"               8 Bits de Poder         ";
320 SYMBOL 111,0,0,&X01111,&X111111,&X111111,&X111111,&X011111,&X01111
330 PRINT CHR$(22)+CHR$(1):' modo transparente
340 i=i+1 :IF i<2 THEN a=2:b=3:c=1: LOCATE 1,1:GOTO 70
350 SYMBOL AFTER 256
351 CALL &BC02:INK 2,7,14:INK 0,0:BORDER 0
360 MEMORY 26999
370 LOAD "!8bp.lib"
380 LOAD "!music.bin"
390 LOAD "!sprites.bin"
400 RUN "!mont7.bas"
```

15.1.2 Código BASIC del juego (mont.bas)

```
10 MEMORY 26999
11 ' esto lo hago antes del defint pues si no dir solo alcanza 32000
12 FOR dir=42580 TO 42618 STEP 2: POKE dir,RND*200:POKE
dir+1,RND*80:NEXT
20 MODE 0: DEFINT A-Z: CALL &6B78:' install RSX
21 ON BREAK GOSUB 1900
22 DIM c$(25):for i=0 to 24:c$(i)=" ":next:' array para layout
23 dim ex(9):dim ey(9): ' arrays para enemigos en pantalla 3 y 4
24 dim exmin(9):dim eymin(9): ' arrays para enemigos en pantalla 3 y 4
25 dim exmax(9):dim eymax(9): ' arrays para enemigos en pantalla 3 y 4
26 llave =&8e7c
50 GOSUB 1000:' presentacion
60 GOSUB 2000:' pantalla 1
70 GOSUB 3000:' pantalla 2
80 gosub 4000:' pantalla 3
90 gosub 5000:' pantalla 4
100 gosub 6000:' pantalla 5
110 GOTO 50:'vuelta a empezar

500 'rutina reseteo de enemigos-----------------------------------
510 FOR i=1 TO 31:|SETUPSP,i,0,&X0:NEXT
```

```
520 RETURN
550 'rutina print layout-----------------------------------------
551 PEN 4:PAPER 7:LOCATE 3,1: PRINT "Mutante Montoya":pen 1:paper 0
560 FOR i=0 TO 23:|LAYOUT,i,0,@c$(i):NEXT
570 RETURN

600 'pantalla de vidas---------------------------------------
610  vidas= vidas-1
611 |MUSICOFF:LOCATE 7,9:PEN 1:PRINT " NIVEL"; nivel
620 LOCATE 7,11:PEN 15:PRINT " VIDAS"; vidas: PEN 1
630 FOR i=1 TO 50: BORDER 7:CALL &BD19:BORDER 0:CALL &BD19:NEXT :CLS
631 vivo=1: IF vidas=0 THEN RUN
640 RETURN

1000 CLS: ' presentacion ------------------------------------
1020 PEN 1:PAPER 0:LOCATE 1,23: PRINT "pulsa S para empezar"
1021 PEN 2:PAPER 0:LOCATE 11,25: PRINT "JJGA 2016"
1030 |MUSICOFF:|MUSIC,1,5
1040 INK 0,0:
1050 |SETUPSP,0,0,&X1:|SETUPSP,0,7,1:|SETUPSP,0,8,2:'status sprite
1051 vidas=5:vivo=1:cs%=0:cl%=0:'variables globales
1070 |SETLIMITS,0,80,0,200:for i=0 to 24:c$(i)=" ":next
1090 c$(0)="                    "
1091 c$(1)=" "
1092 c$(2)="   QQQ"
1100 c$(3)="          QQ"
1110 c$(4)="    V        V"
1120 c$(5)="    X        X QQ"
1130 c$(6)="   QZQQ      Z"
1140 c$(7)="    S        SV"
1150 c$(8)="   XZ        ZX"
1160 c$(9)="   ZZ  X X X ZZ"
1170 c$(10)="   ZZZ Z Z Z ZZ"
1180 c$(11)="   ZZZZZZZZZZZZ"
1190 c$(12)="   ZZZZU TZZZZZ"
1200 c$(13)="   ZSZZ   ZZSSZ"
1210 c$(14)=" W ZZZZ   ZZZZZ    W"
1220 c$(15)=" W ZZZZ   ZWZZZ  WYW"
1230 c$(16)="PYYYYYYYYYYYWYYYYYYW"
1240 c$(17)="PPRRRRRRRRRRRRRRRRRP"
1250 c$(18)="PPYYYWYYYYYYYYWYYYPP"
1260 c$(19)="PPPYWWYYYYWYYYYYYYPP"
1270 GOSUB 550:'print layout
1280 |SETLIMITS,0,80,24,150
1290 |STARS,1,20,2,3,-1
1300 |PRINTSP,0,104,32
1310 IF INKEY(60)=0  THEN RETURN
1320 GOTO 1290

1500 ' rutina movimiento personaje -------------
1510 IF INKEY(27)<0 GOTO 1520
1511 IF INKEY(67)=0 THEN IF dir<>2 THEN |SETUPSP,0,7,2:dir=2:GOTO 1533
ELSE |ANIMA,0:xn=xa+1:yn=ya-2:GOTO 1533
```

```
1512 IF INKEY(69)=0 THEN IF dir<>8 THEN |SETUPSP,0,7,8:dir=8:GOTO 1533
ELSE |ANIMA,0:xn=xa+1:yn=ya+2:GOTO 1533
1513 IF dir<>1 THEN |SETUPSP,0,7,1:dir=1:GOTO 1533 ELSE
|ANIMA,0:xn=xa+1:GOTO 1533
1520 IF INKEY(34)<0 GOTO 1530
1521 IF INKEY(67)=0 THEN IF dir<>4 THEN |SETUPSP,0,7,4:dir=4:GOTO 1533
ELSE |ANIMA,0:xn=xa-1:yn=ya-2:GOTO 1533
1522 IF INKEY(69)=0 THEN IF dir<>6 THEN |SETUPSP,0,7,6:dir=6:GOTO 1533
ELSE |ANIMA,0:xn=xa-1:yn=ya+2:GOTO 1533
1523 IF dir<>5 THEN |SETUPSP,0,7,5:dir=5:GOTO 1533 ELSE
|ANIMA,0:xn=xa-1:GOTO 1533
1530 IF INKEY(67)=0 THEN IF dir<>3 THEN |SETUPSP,0,7,3:dir=3:GOTO 1533
ELSE |ANIMA,0:yn=ya-4:GOTO 1533
1531 IF INKEY(69)=0 THEN IF dir<>7 THEN |SETUPSP,0,7,7:dir=7:GOTO 1533
ELSE |ANIMA,0:yn=ya+4:GOTO 1533
1532 RETURN
1533 |LOCATESP,0,yn,xn:ynn=yn:|COLAY,0,@cl%:IF cl%=0 THEN 1536
1534 yn=ya:|POKE, 27001,yn:|COLAY,0,@cl%:IF cl%=0 THEN 1536
1535 xn=xa: yn=ynn:|POKE, 27001,yn:|POKE, 27003,xn:|COLAY,0,@cl%:IF
cl%=1 THEN yn=ya:|POKE,27001,yn
1536 ya=yn:xa=xn
1537 RETURN

1900 ' rutina on break---------------------------------------
1901 |MUSICOFF
1910 END

2000 'pantalla 1------------------------------------------------
2001 nivel=1:CLS:GOSUB 611
2020 PEN 1:PAPER 0:LOCATE 1,25:PRINT "Navalcarnero cavern"
2030 |MUSICOFF:|MUSIC,0,5
2070 |SETLIMITS,0,80,0,200
2090 c$(1)= "PPPPPPPPPPPPPPPPP  P"
2100 c$(2)= "PU                 P"
2110 c$(3)= "P                  P"
2120 c$(4)= "P                  P"
2130 c$(5)= "P  TPPPPU  TPPPPPPPP"
2140 c$(6)= "P                 TP"
2150 c$(7)= "P                  P"
2160 c$(8)= "P                  P"
2170 c$(9)= "P    YYYYYYYYYY    P"
2190 c$(10)="P    TPPPPPPPPU    P"
2195 c$(11)="P                  P"
2200 c$(12)="P                  P"
2210 c$(13)="P                  P"
2220 c$(14)="YYYYYYYYYP   PYYYYYY"
2230 c$(15)="RRRRRRRRRR   RRRRRRR"
2240 c$(16)="PPPPPPPPPP   PPPPPPP"
2250 c$(17)="PU      TP   PU   TP"
2260 c$(18)="P        T   U     P"
2270 c$(19)="P                  P"
2271 c$(20)="P                  P"
2272 c$(21)="P              W   P"
2273 c$(22)="PP             W  PP"
```

```
2274 c$(23)="PPPPPPPPPPPPPPPPPPPP"
2275 GOSUB 500:'reset enemigos
2280 GOSUB 550:'print layout
2291
e1x=4:e1y=88:e1d=0:|SETUPSP,1,0,&X111:|SETUPSP,1,7,9:|SETUPSP,1,8,0:'s
oldado derecha
2292 e2x=66:e2y=48:e2d=0:|SETUPSP,2,0,&X111:|SETUPSP,2,7,10:'soldado
izq
2293 e3x=30:e3y=16:e3d=1:|SETUPSP,3,0,&X111:|SETUPSP,3,7,10:'soldado
izquierda
2294 |LOCATESP,3,e3y,e3x:|LOCATESP,2,e2y,e2x:|LOCATESP,1,e1y,e1x
2389 xa=8:ya=138:yn=ya:xn=xa:vivo=1:|LOCATESP,0,ya,xa:'"xa" es "x
anterior", "xn" es "x nueva"
2390 GOSUB 1500
2392 IF e1d=0 THEN e1x=e1x+1:IF e1x>=70 THEN e1d=1:|SETUPSP,1,7,10
ELSE 2395
2393 e1x=e1x-1:IF e1x<=4 THEN e1d=0:|SETUPSP,1,7,9
2395 IF e2d=0 THEN e2x=e2x+1:IF e2x>=60 THEN e2d=1:|SETUPSP,2,7,10
ELSE 2398
2396 e2x=e2x-1:IF e2x<=20 THEN e2d=0:|SETUPSP,2,7,9
2398 IF e3d=0 THEN e3x=e3x+1:IF e3x>=70 THEN e3d=1:|SETUPSP,3,7,10
ELSE 2400
2399 e3x=e3x-1:IF e3x<=16 THEN e3d=0:|SETUPSP,3,7,9
2400 POKE 27019,e1x: POKE 27035,e2x:POKE 27051,e3x
2410 |PRINTSPALL,1,0
2411 |COLSP,0,@cs%:IF cs%<32 THEN GOSUB 600:GOTO 2020
2413 IF yn<0 THEN RETURN
2420 GOTO 2390

3000 'pantalla 2--------------------------------------------
3001 nivel=2:CLS:GOSUB 611
3020 PEN 1:PAPER 0:LOCATE 1,25: PRINT " Polvoranca Forrest"
3030 |MUSICOFF:|MUSIC,0,5
3070 |SETLIMITS,0,80,0,200
3090 c$(1)= "OOOOOOOOOOOOOOOOOOOO"
3100 c$(2)= "  W  O             O"
3110 c$(3)= "  W  O             O"
3120 c$(4)= "  W  O             O"
3130 c$(5)= "  W  O     WOOO   OO"
3140 c$(6)= "     O     WOOU    O"
3150 c$(7)= "     O     WOU     O"
3160 c$(8)= "     OOOO  WU      O"
3170 c$(9)= "  W  ORRO  W       O"
3190 c$(10)="  W  ORRO  W       O"
3195 c$(11)="  W  ORRO  W       O"
3200 c$(12)="  W  ORRO  W       O"
3210 c$(13)="  W  ORRO  YYYYYY  O"
3220 c$(14)="  W  YYYY     OOO  O"
3230 c$(15)="  W     O       O  O"
3240 c$(16)="  W     O       O  O"
3250 c$(17)="  W     O OO    O  O"
3260 c$(18)="  W  W  OO  OO  O  O"
3270 c$(19)="  W  O          W   "
3271 c$(20)="  W  O          W   "
```

```
3272 c$(21)="  W  O         W   "
3273 c$(22)="YYYYYYRRRRRRRRRYYYY"
3274 c$(23)="PPPPPPPPPPPPPPPPPPPP"
3275 GOSUB 500:'reset enemigos
3280 GOSUB 550:'print layout
3291
e1x=24:e1y=40:e1d=0:|SETUPSP,1,0,&X111:|SETUPSP,1,7,9:|SETUPSP,1,8,0:'
soldado derecha
3292 e2x=36:e2y=16:e2d=0:|SETUPSP,2,0,&X111:|SETUPSP,2,7,9:'soldado
izq
3293 e3x=0:e3y=16:e3d=0:|SETUPSP,3,0,&X111:|SETUPSP,3,7,11:'fantasma
3294 e4x=24:e4y=166:e4d=0:|SETUPSP,4,0,&X111:|SETUPSP,4,7,12:'pato
3295 e5x=12:e5y=90:e5d=1:|SETUPSP,5,0,&X111:|SETUPSP,5,7,11:'fantasma
3296 e6x=48:e6y=80:e6d=0:|SETUPSP,6,0,&X111:|SETUPSP,6,7,9:'soldado d
3297
|LOCATESP,6,e6y,e6x:|LOCATESP,5,e5y,e5x:|LOCATESP,4,e4y,e4x:|LOCATESP,
3,e3y,e3x:|LOCATESP,2,e2y,e2x:|LOCATESP,1,e1y,e1x
3389 xa=0:ya=152:yn=ya:xn=xa:vivo=1:|LOCATESP,0,ya,xa:'"xa" es "x
anterior", "xn" es "x nueva"
3390 GOSUB 1500
3391 ' logica enemigos
3392 IF e1d=0 THEN e1x=e1x+1:IF e1x>=39 THEN e1d=1:|SETUPSP,1,7,10
ELSE 3395
3393 e1x=e1x-1:IF e1x<=24 THEN e1d=0:|SETUPSP,1,7,9
3395 IF e2d=0 THEN e2x=e2x+1:IF e2x>=70 THEN e2d=1:|SETUPSP,2,7,10
ELSE 3398
3396 e2x=e2x-1:IF e2x<=24 THEN e2d=0:|SETUPSP,2,7,9
3398 IF e3d=0 THEN e3y=e3y+3:IF e3y>=130 THEN e3d=1 ELSE 3410
3399  e3y=e3y-3:IF e3y<=16 THEN e3d=0
3410 IF e4d=0 THEN e4x=e4x+1:IF e4x>=60 THEN e4d=1:|SETUPSP,4,7,13
ELSE 3413
3411 e4x=e4x-1:IF e4x<=24 THEN e4d=0:|SETUPSP,4,7,12
3413 IF e5d=0 THEN e5y=e5y+3:IF e5y>=130 THEN e5d=1 : GOTO 3416 ELSE
3416
3414 e5y=e5y-3:IF e5y<=16 THEN e5d=0
3416 IF e6d=0 THEN e6x=e6x+1:IF e6x>=70 THEN e6d=1:|SETUPSP,6,7,10
ELSE 3418
3417 e6x=e6x-1:IF e6x<=48 THEN e6d=0:|SETUPSP,6,7,9
3418 POKE 27019,e1x: POKE 27035,e2x:POKE 27049,e3y:POKE 27067,e4x:POKE
27081,e5y:POKE 27099,e6x:
3610 |PRINTSPALL,1,0
3611 |COLSP,0,@cs%:IF cs%<32 THEN GOSUB 600:GOTO 3020
3612 IF xn>80 THEN RETURN
3620 GOTO 3390

4000 'pantalla 3-------------------------------------------
4001 nivel=3:CLS:GOSUB 611
4020 PEN 1:PAPER 0:LOCATE 1,25: PRINT "  Valderas castle"
4030 |MUSICOFF:|MUSIC,0,5
4070 |SETLIMITS,0,80,0,200
4090 c$(1)= "ZZZZZZZZZZZZZZZZZZZZ"
4100 c$(2)= "ZSZZSZZZUNTZZSZZZSZZ"
4110 c$(3)= "ZZZZZWZZNNNZZZZZZZZZ"
4120 c$(4)= "ZZZZZWZZNNNZZZZWZZZZ"
```

```
4130 c$(5)= "YYYYYYYYNNNYYYYYYYYY"
4131 c$(6)= "W                  W"
4132 for i=7 to 23 : c$(i)=c$(6):next
4275 GOSUB 500:'reset enemigos
4280 GOSUB 550:'print layout

4390 for i=1 to 4
4391 ex(i)=i*5:ex(i+4)=80-i*5-6
4392 ey(i)=18+i*30:ey(i+4)=18+i*30
4393 |setupsp,i,0,&x11111:|setupsp,i+4,0,&x11111
4394 |setupsp,i,7,9:|setupsp,i+4,7,10
4395 |locatesp,i,ey(i),ex(i):|Locatesp,i+4,ey(i+4),ex(i+4)
4396 |setupsp,i,5,0:|setupsp,i,6,1:|setupsp,i+4,5,0:|setupsp,i+4,6,-1
4399 next
4400 xa=16:ya=200-24-8:yn=ya:xn=xa:vivo=1:|LOCATESP,0,ya,xa
4401 c1=10:c2=20:c=0:cc=0
4402
|SETUPSP,17,9,llave:|SETUPSP,17,0,&x10:|LOCATESP,17,128,72:|printsp,17
,128,72
4409 '----- bucle de logica principal ---------
4410 GOSUB 1500: ' control del personaje
4421 cc=cc+1: if cc=4 then cc=0: gosub 4700
4609 |AUTOALL : ' movimiento automatico de 8 sprites
4610 |PRINTSPALL,1,0 'impresion masiva de sprites (9= 8+personaje)
4611 |COLSP,0,@cs%:IF cs%<32 then if cs%=17 then gosub 4950 else GOSUB
600:GOTO 4020
4612 IF yn<26 THEN RETURN
4620 GOTO 4410
4699 '----- rutinas de control del escuadron ----
4700 c=c+1:if c>c1 and c<=c1+4 then gosub 4800
4701 if c>c2 and c<=c2+4 then gosub 4900
4702 if c=30 then c=10
4710 return
4799 '----- a darse la vuelta de 2 en 2 --------
4800 |setupsp,c1+5-c,7,10:|setupsp,c1+5-c,6,-1
4801 |setupsp,c1+5-c+4,7,9:|setupsp,c1+5-c+4,6,1
4810 return
4899 '----- a darse la vuelta otra vez --------
4900 |setupsp,c2+5-c,7,9:|setupsp,c2+5-c,6,1
4901 |setupsp,c2+5-c+4,7,10:|setupsp,c2+5-c+4,6,-1
4910 return
4949 '------- apertura del castillo -----------
4950 c$(2)= "UM"
4952 c$(3)= "MMM"
4953 c$(4)= "MMM"
4954 c$(5)= "MMM"
4955 for i=2 to 5:|LAYOUT,i,8,@c$(i):NEXT
4956 |SETUPSP,17,0,&x00
4957 c$(2)= "U "
4958 c$(3)= "   "
4959 c$(4)= "   "
4960 c$(5)= "   "
4961 for i=2 to 5:|LAYOUT,i,8,@c$(i):NEXT
4962 return
```

```
4999 'pantalla 4-------------------------------------------
5000 nivel=4:CLS:GOSUB 611
5020 PEN 2:PAPER 0:LOCATE 1,25: PRINT "   Falken's maze"
5030 |MUSICOFF:|MUSIC,0,5
5040 |SETLIMITS,0,80,0,200
5041 ' inicializacion de laberinto y enemigos
5042 GOSUB 500:'reset enemigos
5043 t=0:' contador de sprites y tiempo
5050 c$(1)=  "ZZNNZZNNZZNNZZNNZZZZ"
5051 c$(2)=  "Z                  Z"
5052 c$(3)=  "Z                  Z"
5053 c$(4)=  "Z                  S"
5054 c$(5)=  "S               X  Z"
5055 c$(6)=  "Z               Z  Z"
5056 c$(7)=  "Z               S  Z"
5057 c$(8)=  "Z               Z  S"
5058 c$(9)=  "Z               Z  Z"
5059 c$(10)= "Z               Z  Z"
5060 c$(11)= "Z  X  ZZZZZZZZZZU  Z"
5061 c$(12)= "Z  Z  N            Z"
5062 c$(13)= "Z  S  N            Z"
5063 c$(14)= "Z  Z  N            Z"
5064 c$(15)= "Z  T  ZZZZU  TZZZZZZ"
5065 c$(16)= "Z                  Z"
5066 c$(17)= "Z                  S"
5067 c$(18)= "Z                  Z"
5068 c$(19)= "Z  TZZZZZUXTZZZZZ  Z"
5069 c$(20)= "Z         S        Z"
5070 c$(21)= "Z         Z        Z"
5071 c$(22)= "Z         Z        Z"
5072 c$(23)= "TZZNNZZZ  Z  ZZNNZZU"
5073 GOSUB 550: 'layout
5080 for i=1 to 6:|setupsp,i,0,&x0:|setupsp,i,7,11:next:'fantasmas
5081 xa=32:ya=168:yn=ya:xn=xa:|LOCATESP,0,yn,xn: ' personaje
5082 'enemigo 1---------
5084 |setupsp,1,0,&x01111:ey(1)=30:ex(1)=10
5085 |setupsp,1,5,-2:|setupsp,1,6,1:incy(1)=2
5086 |locatesp,1,ey(1),ex(1):exmax(1)=50:exmin(1)=8:eymax(1)=56:eymin(1)=22
5090 'enemigo 2---------
5091 |setupsp,2,0,&x01111:ey(2)=42:ex(2)=30
5092 |setupsp,2,5,-2:|setupsp,2,6,1: incy(2)=2
5093 |locatesp,2,ey(2),ex(2):exmax(2)=50:exmin(2)=8:eymax(2)=56:eymin(2)=22
5100 'enemigo 3---------
5101 |setupsp,3,0,&x01111:ey(3)=27:ex(3)=4
5102 |setupsp,3,5,3:|setupsp,3,6,0: incy(3)=3
5103 |locatesp,3,ey(3),ex(3):exmax(3)=80:exmin(3)=0:eymax(3)=150:eymin(3)=22
5110 'enemigo 4---------
5111 |setupsp,4,0,&x01111:ey(4)=12*8+1:ex(4)=34
```

```
5112 |setupsp,4,5,0:|setupsp,4,6,1: incy(4)=0
5113 |locatesp,4,ey(4),ex(4):exmax(4)=61:exmin(4)=34:eymax(4)=200:eymin(4)=0
5120 'enemigo 5---------
5121 |setupsp,5,0,&x01111:ey(5)=16*8+1:ex(5)=60
5122 |setupsp,5,5,0:|setupsp,5,6,-1: incy(5)=0
5123 |locatesp,5,ey(5),ex(5):exmax(5)=63:exmin(5)=25:eymax(5)=200:eymin(5)=0

5500 GOSUB 1500: ' control del personaje
5510 t=t+1: gosub 5700
5511 if t=5 then t=0
5520 |AUTOALL : ' movimiento automatico de 8 sprites
5530 |PRINTSPALL,1,0 'impresion masiva de sprites (9= 8+personaje)
5540 |COLSP,0,@cs%:IF cs%<32 then GOSUB 600:GOTO 5020
5550 if xn>40 and yn>=168 then return
5560 GOTO 5500

5699'logica generica. solo se ejecuta la del sprite "t"
5700 '
5701 ex(t)=peek (27003+16*t)
5710 if ex(t)>exmax(t) then |setupsp,t,6,-1:   else if  ex(t)<exmin(t) then |setupsp,t,6,1
5720 if incy(t)=0 then return
5721 ey(t)=peek (27001+16*t)
5730 if ey(t)>eymax(t) then |setupsp,t,5,-incy(t):   else if ey(t)<eymin(t) then |setupsp,t,5,incy(t)
5790 return


5999' 'pantalla 5 ------------------------------------------
6000 nivel=5:CLS:GOSUB 611
6020 PEN 2:PAPER 0:LOCATE 1,25: PRINT " Rescue princess"
6030 |MUSICOFF:|MUSIC,2,6
6040 |SETLIMITS,0,80,0,200
6041 ' inicializacion de laberinto y enemigos
6042 GOSUB 500:'reset enemigos
6043 t=0:' contador de sprites y tiempo
6050 c$(1)=  "                    "
6051 c$(2)=  "QQQ  V        V     "
6052 c$(3)=  "     X        X QQQQ"
6053 c$(4)=  "    XZ        Z  QQQ"
6054 c$(5)=  "    ZZ   Z Z ZZV    "
6055 c$(6)=  "    SZNNZZZZZZZX    "
6056 c$(7)=  "    ZZ        ZS  QQ"
6057 c$(8)=  "    TZ        ZZ    "
6058 c$(9)=  "QQ   Z        ZU V  "
6059 c$(10)= "   QQZZZZ  ZZZZ  X  "
6060 c$(11)= "     Z           S  "
6061 c$(12)= "  V  Z           Z  "
6062 c$(13)= "  X  Z           Z  "
6063 c$(14)= " QS  ZZZZ  ZZZZZZU  "
```

```
6064 c$(15)= "  Z        Z  ZU    "
6065 c$(16)= "  Z        Z  Z X QQ"
6066 c$(17)= "  Z        Z  Z Z   "
6067 c$(18)= "  TZZZZZZ  ZNNZ Z   "
6068 c$(19)= "    TZU T     Z Z   "
6069 c$(20)= "     Z        ZZZ   "
6070 c$(21)= "  QQQZ        ZU  W "
6071 c$(22)= "     Z        Z   W "
6072 c$(23)= "YYYYYZ        ZYYYYY"
6073 GOSUB 550: 'layout
6074
|SETUPSP,17,9,llave:|SETUPSP,17,0,&x10:|LOCATESP,17,15*8,13*4:|printsp
,17,15*8,13*4:'llave1
6075
|SETUPSP,16,9,llave:|SETUPSP,16,0,&x10:|LOCATESP,16,9*8,6*4:|printsp,1
6,9*8,6*4:'llave2
6076 |SETUPSP,15,7,15:|SETUPSP,15,0,&x111:|LOCATESP,15,3*8,12*4:'
princess


6291 e1x=28:e1y=7*8:e1d=0:|SETUPSP,1,0,&X111:|SETUPSP,1,7,9:'soldado
derecha
6292 e2x=51:e2y=11*8:e2d=1:|SETUPSP,2,0,&X111:|SETUPSP,2,7,10:'soldado
izq
6293 e3x=30:e3y=15*8:e3d=1:|SETUPSP,3,0,&X111:|SETUPSP,3,7,10:'soldado
izquierda
6294 |LOCATESP,3,e3y,e3x:|LOCATESP,2,e2y,e2x:|LOCATESP,1,e1y,e1x

6389
princes=0:xa=28:ya=21*8:yn=ya:xn=xa:vivo=1:|LOCATESP,0,ya,xa:'"xa" es
"x anterior", "xn" es "x nueva"
6390 GOSUB 1500
6392 IF e1d=0 THEN e1x=e1x+1:IF e1x>=51 THEN e1d=1:|SETUPSP,1,7,10
ELSE 6395
6393 e1x=e1x-1:IF e1x<=28 THEN e1d=0:|SETUPSP,1,7,9
6395 IF e2d=0 THEN e2x=e2x+1:IF e2x>=61 THEN e2d=1:|SETUPSP,2,7,10
ELSE 6398:'51
6396 e2x=e2x-1:IF e2x<=24 THEN e2d=0:|SETUPSP,2,7,9
6398 IF e3d=0 THEN e3x=e3x+1:|ANIMA,3:IF e3x>=38 THEN
e3d=1:|SETUPSP,3,7,10 ELSE 6400
6399 e3x=e3x-1:|ANIMA,3:IF e3x<=12 THEN e3d=0:|SETUPSP,3,7,9
6400 POKE 27019,e1x: POKE 27035,e2x:POKE 27051,e3x
6410 |PRINTSPALL,1,0
6411 |COLSP,0,@cs%:IF cs%<32 THEN IF cs%>=15 then gosub 6500 else
GOSUB 600:GOTO 6020
6413 IF princes=1 THEN 6600
6420 GOTO 6390

6499' llave1, llave 2 y princesa
6500 borra$="MM":spaces$="  "
6501 if cs%=16 then
|LAYOUT,18,12,@borra$:|LAYOUT,18,12,@spaces$:|SETUPSP,16,0,&x0:return
6510 if cs%=17 then
|LAYOUT,6,6,@borra$:|LAYOUT,6,6,@spaces$:|SETUPSP,17,0,&x0:return
```

```
6520 if cs%=15 then princes=1
6530 return

6600' MISION CUMPLIDA
6610 |MUSICOFF:|MUSIC,2,8


6630 ink 14,0 :paper 14:pen 15
6640 locate 1,22: print "   Felicidades!     ":pen 1
6650 locate 1,23: print " Has logrado raptar "
6655 locate 1,24: print "    a la princesa!  "
6660 locate 1,25: print "Pide un buen rescate";
6670 paper 0:
6680 border 7:FOR i=1 TO 30000:NEXT
6681 gosub 7000
6690 return




7000' 'pantalla end of game  -----------------------------------------
---
7020 cls:border 0:PEN 2:PAPER 0:LOCATE 1,25: PRINT "  8 bits de poder!
"
7030 |MUSICOFF:|MUSIC,2,8
7040 |SETLIMITS,0,80,0,200
7042 GOSUB 500:'reset enemigos
7050 c$(1)=  "ZZZ              ZZZ"
7051 for i=2 to 21: c$(i)=c$(1):next
7071 c$(22)= "YYYYYYYYYYYYYYYYYYYY"
7072 c$(23)= "OOOOOOOOOOOOOOOOOOOO"
7073 GOSUB 550: 'layout
7076
|SETUPSP,15,7,15:|SETUPSP,15,0,&x111:|LOCATESP,15,19*8,11*4:'princesa
7077 |SETUPSP,2,7,10:|SETUPSP,2,0,&x111:|LOCATESP,2,19*8,13*4:'soldado
izq
7078 |SETUPSP,3,7,11:|SETUPSP,3,0,&x111:|LOCATESP,3,19*8,7*4:'
monstruo
7079 |SETUPSP,4,7,12:|SETUPSP,4,0,&x111:|LOCATESP,4,21*8,4*4:' pato
7080 |SETUPSP,0,7,7:|SETUPSP,0,0,&x111:|LOCATESP,0,19*8,9*4:' montoya
down
7081 |SETUPSP,5,7,9:|SETUPSP,5,0,&x111:|LOCATESP,5,19*8,5*4:'soldado
izq
7082 |SETUPSP,6,7,13:|SETUPSP,6,0,&x111:|LOCATESP,6,21*8,15*4:' pato
7089 telon=&8ec0
7090 |setupsp,1,9,telon
7100 for y=8 to 168 step 4
7110 for x=12 to 64 step 4
7111 |PRINTSP,1,y,x
7112 next
7113 next
7114 for y=168 to 100 step -2
7115 for x=12 to 64 step 4
7116 |PRINTSP,1,y,x
```

```
7117 next
7118 next:LOCATE 5,16:pen 3: PRINT "S para salir"
7119 IF INKEY(60)=0  THEN RETURN
7190 |PRINTSPALL,1,1:for j=1 to 100:next:goto 7119
```

15.2 Ficheros asm

15.2.1 Fichero de ensamblado global (make_all.asm)

```
; Makefile para los videojuegos que usan 8bits de poder
; si alteras solo una parte solo tienes que ensamblar el make
correspondiente
; por ejemplo puedes ensamblar el make_graficos si cambias dibujos
;--------------------CODIGO ------------------------------
;incluye la libreria 8bp y el playerWYZ de musica
read "make_codigo.asm"

;--------------------MUSICA-----------------------------------
; incluye las canciones.
read "make_musica.asm"

; ---------- GRAFICOS -------------------
; esta parte incluye imagenes y secuencias de animacion
; y la tabla de sprites inicializada con dichas imagenes y secuencias
read "make_graficos.asm"
```

15.2.2 Fichero de graficos (make_graficos.asm)

```
; ---------- datos de secuencias e imagenes -------------------
; aqui se almacenan tanto las secuencias como las imagenes
; las secuencias se pueden hacer en basic con SETUPSQ o bien
; mediante POKE


org 33500
read "sequences_montoya.asm"

org 34000
read "images_montoya.asm"
;---------------------TABLA DE SPRITES-------------------------
; esta parte es opcional. puedes manipular la tabla desde basic
; usando SETUPSP y LOCATESP o bien directamente con POKE
; aunque con POKE no puedes insertar coordenadas negativas

org 27000
read "sprites_table.asm"
```

15.2.3 Fichero de secuencias (sequences_montoya.asm)

```
org 33500;
;========================================================================
; 31 secuencias de animacion (500bytes) de 8 frames
;========================================================================
; debe ser una tabla fija y no variable
; cada secuencia contiene las direcciones de frames de animacion
ciclica
; cada secuencia son 8 direcciones de memoria de imagen
; numero par porque las animaciones suelen ser un numero par
; un cero significa fin de secuencia, aunque siempre se
; gastan 8 words /secuencia
; al encontrar un cero se comienza de nuevo.
; si no hay cero, tras el frame 8 se comienza de nuevo
; en total caben 31 secuencias diferentes (disponemos de 500 bytes)
; SEQUENCES:
; la secuencia cero es que no hay secuencia.
; empezamos desde la secuencia 1

;--------------secuencias de animacion del personaje --------
dw MONTOYA_R0,MONTOYA_R1,MONTOYA_R2,MONTOYA_R1,0,0,0,0       ;1
dw MONTOYA_UR0,MONTOYA_UR1,MONTOYA_UR2,MONTOYA_UR1,0,0,0,0  ;2
dw MONTOYA_U0,MONTOYA_U1,MONTOYA_U0,MONTOYA_U2,0,0,0,0      ;3
dw MONTOYA_UL0,MONTOYA_UL1,MONTOYA_UL2,MONTOYA_UL1,0,0,0,0  ;4
dw MONTOYA_L0,MONTOYA_L1,MONTOYA_L2,MONTOYA_L1,0,0,0,0      ;5
dw MONTOYA_DL0,MONTOYA_DL1,MONTOYA_DL2,MONTOYA_DL1,0,0,0,0  ;6
dw MONTOYA_D0,MONTOYA_D1,MONTOYA_D0,MONTOYA_D2,0,0,0,0      ;7
dw MONTOYA_DR0,MONTOYA_DR1,MONTOYA_DR2,MONTOYA_DR1,0,0,0,0  ;8

;--------------secuencias de animacion del soldado --------
dw SOLDADO_R0,SOLDADO_R2,SOLDADO_R1,SOLDADO_R2,0,0,0,0 ;9
dw SOLDADO_L0,SOLDADO_L2,SOLDADO_L1,SOLDADO_L2,0,0,0,0 ;10

;--------------secuencias de animacion del fantasma --------
dw FANTASMA_0,FANTASMA_1,FANTASMA_2,0,0,0,0,0;11

;--------------------------------------pato
dw PATO_D,0,0,0,0,0,0,0;12
dw PATO_I,0,0,0,0,0,0,0;13
dw NAVE_MALA,0,0,0,0,0,0,0;14 ojo no lo uso
;-----------------------------------princesa
PRINCESS,PRINCESS,PRINCESS,PRINCESS,PRINCESS2,PRINCESS2,PRINCESS2,PRIN
CESS2;15
```

15.2.4 Fichero de musica (make_musica.asm)

```
; si compilas esto independientemente
; deberia ser al menos donde acaba el codigo de 8bp y del player,
comprobando
; donde se ensambla la etiqueta _FIN_CODIGO.
; suponiendo que es menor de 32000 (en realidad es algo menos, puedes
ensamblar en 32000)
; tras ensamblarlo, salvalo con save "musica.bin",b,32000,1500
```

```
org 32050
;--------------------MUSICA------------------------------------
; tiene la limitacion de tan solo poder incluir un solo fichero de
; instrumentos para todas las canciones
; la limitacion se solventa simplemente metiendo todos los
; instrumentos en un solo fichero.

;archivo de instrumentos. OJO TIENE QUE SER SOLO UNO
read  "montoya7.mus.asm"
;read "medieval5.mus.asm" ;
;read "pingu4.mus.asm"

; archivos de musica
; ojo la primera nota debe sonar en los 3 canales y ademas ya nunca se
repetira
; si no hacemos sonar una primera nota, el canal se queda mudo. Parece
un bug del player aunque
; no tiene ningun efecto negativo si cumplimos esta regla.
; IMPORTANTE esta nota especial debe ser del instrumento con id=0
(edito usando WYZ tracker)
; si es de otro instrumento me da problemas.
SONG_0:
INCBIN      "montoya7.mus" ;
SONG_0_END:



SONG_1:
INCBIN      "pingu4.mus" ;
SONG_1_END:

SONG_2:
INCBIN      "medieval5.mus" ;
SONG_2_END:
SONG_3:
SONG_4:
SONG_5:
SONG_6:
SONG_7:
```

15.2.5 Fichero de tabla de sprites (sprites_table.asm)

```
org 27000
;------- sprite 0 MONTOYA------------------
sprite0     db 1; status, es el byte de flags

            dw 50; coordy
            dw 50; coordx
            db 0; velocidadY para movimiento automatico
            db 0; velocidadX para movimiento automatico
            db 2; identificador de secuencia de animacion
   ; un 0 significa que no hay secuencia de animacion
            db 0; identificador de frame en la secuencia de
animacion[0..n]
```

```
            ; si sa secuencia de animacion es cero , este campo se
ignora
            dw MONTOYA_D0; direccion de la imagen 0
            ; si hay una secuencia  de animacion asignada, entonces
este
;dato es redundante con el id_frame. Aun asi es necesario.
            ; Ademas si no hay secuencia es la unica forma de indicar
cual
;es la imagen. ejemplo hierba, ladrillos, etc objetos sin
;secuencias de animacion

            ds 5 ; posible uso futuro
;---------------------------------------------------------------------
sprite1     ds 16;
sprite2     ds 16;
sprite3     ds 16;
sprite4     ds 16;
sprite5     ds 16;
sprite6     ds 16;
sprite7     ds 16;
sprite8     ds 16;
sprite9     ds 16;
sprite10    ds 16;
sprite11    ds 16;
sprite12    ds 16;
sprite13    ds 16;
sprite14    ds 16;
sprite15    ds 16;
sprite16    ds 16;
sprite17    ds 16;
;------------------------------------------------------------
; AQUI METO LOS LADRILLOS. 14 en total (14 sprites)
; sprite 18 --> M void (cuadro todo a ceros)
; sprite 19 --> N rejas
; sprite 20 --> O hojas
; sprite 21 --> P roca
; sprite 22 --> Q nube
; sprite 23 --> R agua
; sprite 24 --> S ventana
; sprite 25 --> T arco R
; sprite 26 --> U arco L
; sprite 27 --> V bandera
; sprite 28 --> W planta
; sprite 29 --> X picotorre
; sprite 30 --> Y cesped
; sprite 31 --> Z ladrillo
;------------------------------------------------------------ M
sprite18
      db 0
      dw 0; y
      dw 0; x
      db 0; vy
      db 0;vx
      db 0;
```

```
      db 0;
      dw VOID
      ds 5 ; posible uso futuro
;----------------------------------------------------------- N
sprite19
      db 0
      dw 0; y
      dw 0; x
      db 0; vy
      db 0;vx
      db 0;
      db 0;
      dw REJA
      ds 5 ; posible uso futuro
;----------------------------------------------------------- O
sprite20
      db 0
      dw 0; y
      dw 0; x
      db 0; vy
      db 0;vx
      db 0;
      db 0;
      dw HOJAS
      ds 5 ; posible uso futuro
;----------------------------------------------------------- P
sprite21
      db 0
      dw 0; y
      dw 0; x
      db 0; vy
      db 0;vx
      db 0;
      db 0;
      dw ROCA
      ds 5 ; posible uso futuro
;----------------------------------------------------------- Q
sprite22
      db 0
      dw 0; y
      dw 0; x
      db 0; vy
      db 0;vx
      db 0;
      db 0;
      dw NUBE
      ds 5 ; posible uso futuro
;----------------------------------------------------------- R
sprite23
      db 0
      dw 0; y
      dw 0; x
      db 0; vy
      db 0;vx
```

```
      db 0;
      db 0;
      dw AGUA
      ds 5 ; posible uso futuro
;---------------------------------------------------------- S
sprite24
      db 0
      dw 0; y
      dw 0; x
      db 0; vy
      db 0;vx
      db 0;
      db 0;
      dw VENTANA
      ds 5 ; posible uso futuro
;---------------------------------------------------------- T
sprite25
      db 0
      dw 0; y
      dw 0; x
      db 0; vy
      db 0;vx
      db 0;
      db 0;
      dw ARCOR
      ds 5 ; posible uso futuro
;---------------------------------------------------------- U
sprite26
      db 0
      dw 0; y
      dw 0; x
      db 0; vy
      db 0;vx
      db 0;
      db 0;
      dw ARCOL
      ds 5 ; posible uso futuro
;---------------------------------------------------------- V
sprite27
      db 0
      dw 0; y
      dw 0; x
      db 0; vy
      db 0;vx
      db 0;
      db 0;
      dw BANDERA
      ds 5 ; posible uso futuro
;---------------------------------------------------------- W
sprite28
      db 0
      dw 0; y
      dw 0; x
      db 0; vy
```

```
      db 0;vx
      db 0;
      db 0;
      dw PLANTA
      ds 5 ; posible uso futuro
;------------------------------------------------------------ X
sprite29
      db 0
      dw 0; y
      dw 0; x
      db 0; vy
      db 0;vx
      db 0;
      db 0;
      dw PICOTORRE
      ds 5 ; posible uso futuro
;------------------------------------------------------------ Y
sprite30
      db 0
      dw 0; y
      dw 0; x
      db 0; vy
      db 0;vx
      db 0;
      db 0;
      dw CESPED
      ds 5 ; posible uso futuro
;----------------------------------------------------- Z SPRITE NUMERO
31
sprite31
      db 0
      dw 0; y
      dw 0; x
      db 0; vy
      db 0;vx
      db 0;
      db 0;
      dw LADRILLO
      ds 5 ; posible uso futuro
```

16 APENDICE II: Organización de la memoria de video

16.1 El ojo humano y la resolución del CPC

La memoria de video del amstrad CPC tiene 3 modos de funcionamiento. El modo mas utilizado para los juegos es el mode 0 (160x200) por disponer de mas color pero tambien se ha usado bastante el mode 1 (320x200) para programar juegos.

Puesto que la cantidad de memoria de video es la misma, se sacrifica resolución para ganar en cantidad de colores, pero curiosamente en horizontal, que es el lado de la pantalla mas largo, hay menos resolución que en vertical (160 en horizontal y 200 en vertical). Te preguntarás por qué. Además no es el unico microordenador que hacía eso, muchos otros ordenadores también usaban la misma estrategia con el lado horizontal

El motivo tiene que ver con el funcionamiento del sistema visual humano. El ojo percibe mas detalles en vertical que en horizontal, de modo que "dañar" la resolución en el eje horizontal no es tan grave como dañarla en vertical. Subjetivamente es mas aceptable el resultado.

16.2 La memoria de video

La información mas completa y clara se encuentra en el manual del firmware del amstrad. Esta información te será util si quieres construir un editor de sprites mejorado o quieres adentrarte en el ensamblador y programar rutinas de impresión con sobreescritura o cualquier otra cosa.

16.2.1 Mode 2

En mode 2, cada píxel esta representado por un bit. De modo que un byte representa a 8 pixels. Si tomamos cualquier byte de la memoria de video, su correspondencia con los pixeles es de 1 bit por cada píxel, en esta tabla se representan los bits y a que pixeles (pi) corresponden

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
p0	p1	p2	p3	p4	p5	p6	p7

en un byte se numera el bit 7 como el mas situado a la izquierda. El píxel 0, es precisamente tambien el pixel situado mas a la izquierda, es decir, aquí no hay nada "al reves". Esta todo correcto

16.2.2 Mode 1

En mode 1 tenemos 4 colores (representados por 2 bits). Un byte representa por lo tanto a 4 pixeles. La correspondencia entre pixeles y bits es algo mas compleja. El pixel 0 por ejemplo se codifica con los bits 7 y 4

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
p0(1)	p1(1)	p2(1)	p3(1)	p0(0)	p1(0)	p2(0)	p3(0)

16.2.3 Mode 0

Aquí tenemos un pequeó follón. Cada byte representa solo dos pixels, de los cuales la correspondencia con los bits del byte es la siguiente: El pixel 0 se codifica con los bits 7,5,3 y 1

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
p0(0)	p1(0)	p0(2)	p1(2)	p0(1)	p1(1)	p0(3)	p1(3)

16.2.4 Lineas de la pantalla

Los pixels de la pantalla pertenecientes a una misma linea se encuentran codificados en los bytes que tambien son contiguos. Sin embargo, de una linea a otra hay saltos.
Si avanzamos en direcciones de memoria, al llegar al final de una linea saltamos a una linea que se encuentra 8 lineas mas abajo. Y si queremos continuar en la linea siguiente lo que tenemos que saltar en direcciones de memoria son 2048 posiciones

LINE	R0W0	R0W1	R0W2	R0W3	R0W4	R0W5	R0W6	R0W7
1	C000	C800	D000	D800	E000	E800	F000	F800
2	C050	C850	D050	D850	E050	E850	F050	F850
3	C0A0	C8A0	D0A0	D8A0	E0A0	E8A0	F0A0	F8A0
4	C0F0	C8F0	D0F0	D8F0	E0F0	E8F0	F0F0	F8F0
5	C140	C940	D140	D940	E140	E940	F140	F940
6	C190	C990	D190	D990	E190	E990	F190	F990
7	C1E0	C9E0	D1E0	D9E0	E1E0	E9E0	F1E0	F9E0
8	C230	CA30	D230	DA30	E230	EA30	F230	FA30
9	C280	CA80	D280	DA80	E280	EA80	F280	FA80
10	C2D0	CAD0	D2D0	DAD0	E2D0	EAD0	F2D0	FAD0
11	C320	CB20	D320	DB20	E320	EB20	F320	FB20
12	C370	CB70	D370	DB70	E370	EB70	F370	FB70
13	C3C0	CBC0	D3C0	DBC0	E3C0	EBC0	F3C0	FBC0
14	C410	CC10	D410	DC10	E410	EC10	F410	FC10
15	C460	CC60	D460	DC60	E460	EC60	F460	FC60
16	C4B0	CCB0	D4B0	DCB0	E4B0	ECB0	F4B0	FCB0
17	C500	CD00	D500	DD00	E500	ED00	F500	FD00
18	C550	CD50	D550	DD50	E550	ED50	F550	FD50
19	C5A0	CDA0	D5A0	DDA0	E5A0	EDA0	F5A0	FDA0
20	C5F0	CDF0	D5F0	DDF0	E5F0	ED50	F550	FD50
21	C640	CE40	D640	DE40	E640	EE40	F640	FE40
22	C690	CE90	D690	DE90	E690	EE90	F690	FE90
23	C6E0	CEE0	D6E0	DEE0	E6E0	EEE0	F6E0	FEE0
24	C730	CF30	D730	DF30	E730	EF30	F730	FF30
25	C780	CF80	D780	DF80	E780	EF80	F780	FF80
spare start	C7D0	CFD0	D7D0	DFD0	E7D0	EFD0	F7D0	FFD0
spare end	C7FF	CFFF	D7FF	DFFF	E7FF	EFFF	F7FF	FFFF

Fig. 35 Mapa de memoria de pantalla

16.3 Cómo hacer una pantalla de carga para tu juego

Hay muchas formas de hacerla. Una muy sencilla es construir un gráfico con el programa spedit modificado para que te permita pintar en toda la pantalla sin mostrarte menus y al final pulsar alguna tecla que te lance un comando SAVE como este

SAVE “mipantalla.bin”,b,&C000,16384

Como ves el comando salva 16KB desde la dirección de comienzo de la pantalla, que es la &C000
La forma de cargarlo sería

LOAD “mipantalla.bin”,&C000

Y verías poco a poco como mientras carga se va dibujando en pantalla, puesto que es ahí precisamente donde lo estas cargando, en la memoria de video.

Otra forma es generar un layout bien trabajado y salvarlo mediante el comando SAVE anterior

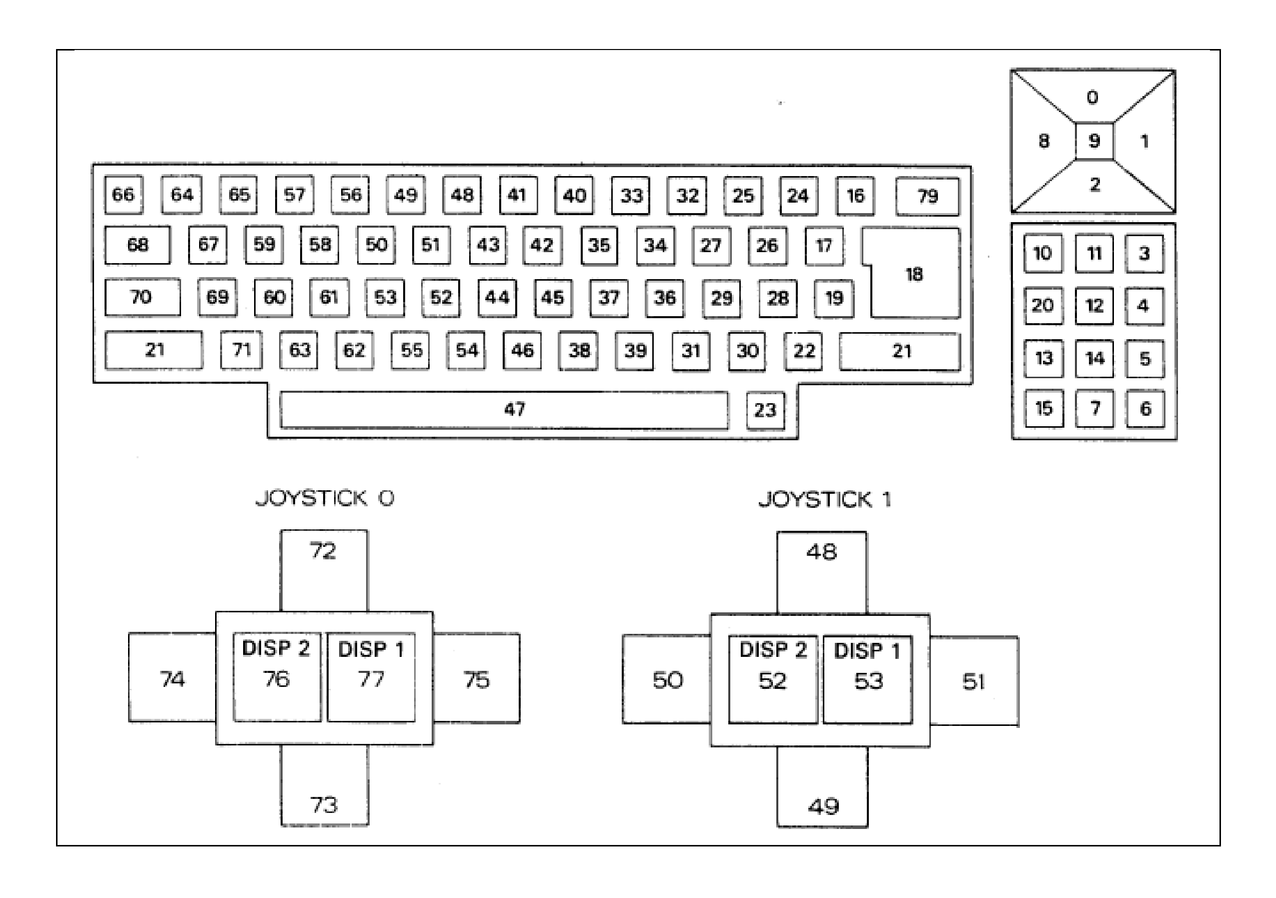
Por ultimo puedes usar una herramienta como ConvImgCPC (tambien hay otras), un conversor de imágenes que funciona bajo windows. Esta herramienta te permite transformar una imagen cualquiera (que puede ser un escaneo de un dibujo tuyo) en un fichero binario (con extensión .scr) apto para el CPC

Para meter este fichero en un disco (en un fichero .dsk) debes usar el CPCDiskXP que es otra herramienta que te permite meter ficheros dentro de ficheros .dsk

Una vez dentro del .dsk puedes cargarlo con LOAD “mipantalla.bin”,&C000
Sin embargo los colores no se veran bien porque ConvImgCPC ajusta la paleta para aproximarse lo mas posible a los colores originales. Si invocas CALL &C7D0 conseguiras ver la imagen con los colores seleccionados por convImgCPC

Para dejar la paleta en sus valores por defecto, usa CALL &BC02, que es una rutina del firmware.

17 APENDICE III: INKEY codes



18 APENDICE IV: Paleta

Los 27 colores son:

0 - Negro (5)	1 - Azul (0,14)	2 - Azul claro (6)	3 - Rojo	4 - Magenta	5 - Violeta	6 - Rojo claro (3)	7 - Púrpura	8 - Magenta claro (7)
9 - Verde	10 - Cyan (8)	11 - Azul cielo (15)	12 - Amarillo (9)	13 - Gris	14 - Azul pálido (10)	15 - Anaranjado	16 - Rosa (11)	17 - Magenta pálido
18 - Verde claro (12)	19 - Verde mar	20 - Cyan claro (2)	21 - Verde lima	22 - Verde pálido (13)	23 - Cyan pálido	24 - Amarillo claro (1)	25 - Amarillo pálido	26 - Blanco (4)

Los valores de la paleta por defecto en cada modo son:



Modo 2:

0: Azul (paleta 1)	1: Amarillo intenso (paleta 24)

Modo 1:

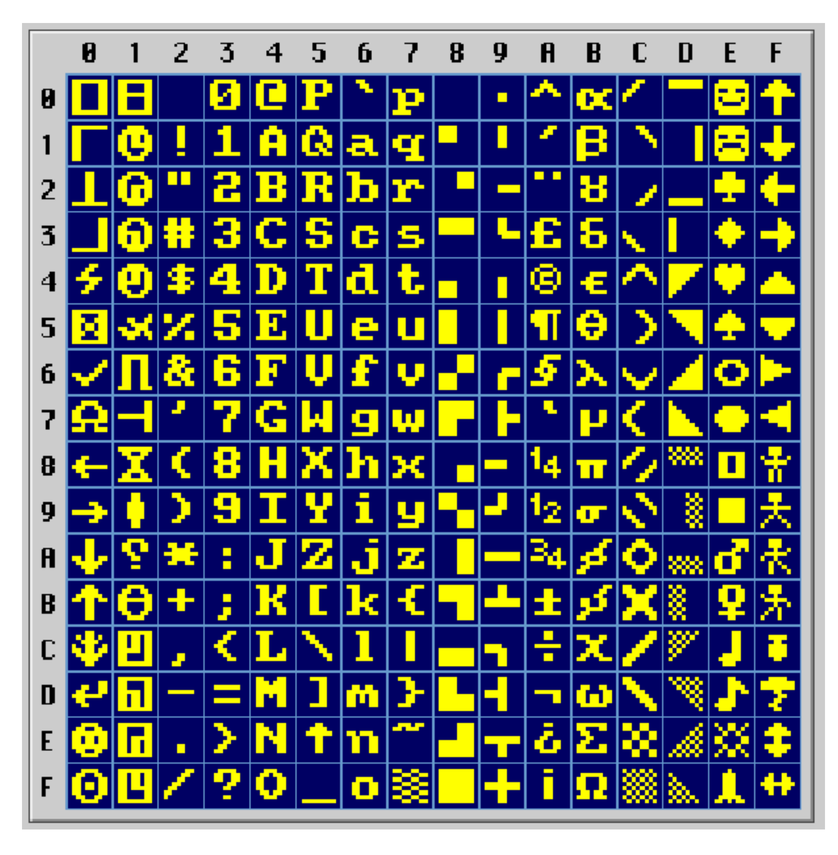
0: Azul (paleta 1)	1: Amarillo intenso (paleta 24)
2: Cyan claro (paleta 20)	3: Rojo claro (paleta 6)

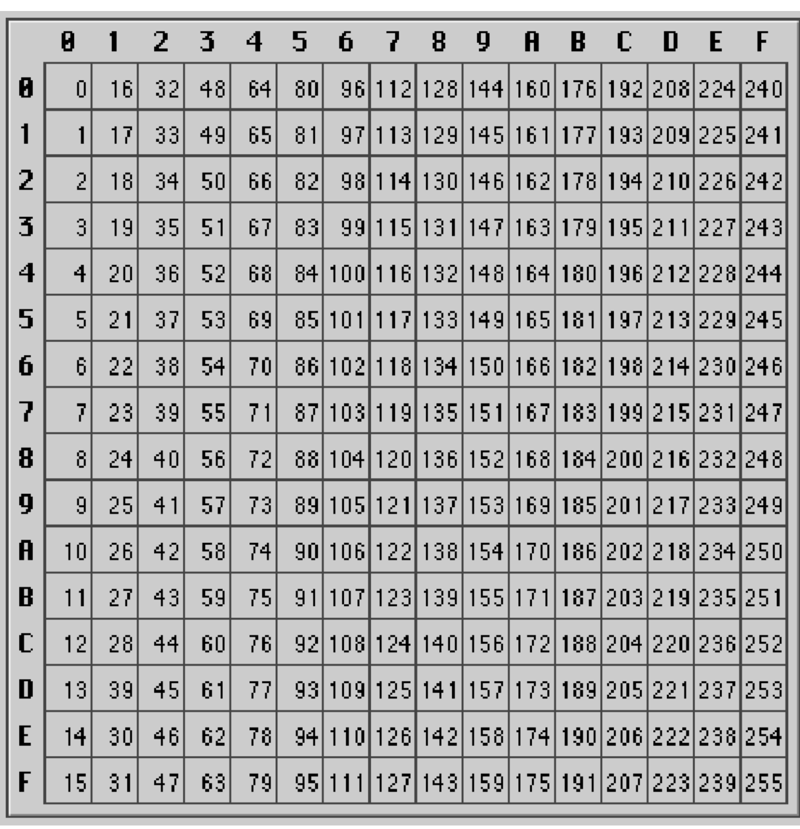
Modo 0:

0: Azul (paleta 1)	1: Amarillo intenso (paleta 24)	2: Cyan claro (paleta 20)	3: Rojo claro (paleta 6)
4: Blanco (paleta 26)	5: Negro (paleta 0)	6: Azul claro (paleta 2)	7: Magenta claro (paleta 8)
8: Cyan (paleta 10)	9: Amarillo (paleta 12)	10: Azul pálido (paleta 14)	11: Rosa (paleta 16)
12: Verde claro (paleta 18)	13: Verde pálido (paleta 22)	14: Parpadeo Azul/Amarillo	15: Parpadeo azul cielo/Rosa

Los valores de la paleta en cada modo se gestionan con el comando INK, consulta el manual de referencia BASIC del amstrad para más información.

19 APENDICE V: Tabla ASCII del AMSTRAD CPC





	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240
1	1	17	33	49	65	81	97	113	129	145	161	177	193	209	225	241
2	2	18	34	50	66	82	98	114	130	146	162	178	194	210	226	242
3	3	19	35	51	67	83	99	115	131	147	163	179	195	211	227	243
4	4	20	36	52	68	84	100	116	132	148	164	180	196	212	228	244
5	5	21	37	53	69	85	101	117	133	149	165	181	197	213	229	245
6	6	22	38	54	70	86	102	118	134	150	166	182	198	214	230	246
7	7	23	39	55	71	87	103	119	135	151	167	183	199	215	231	247
8	8	24	40	56	72	88	104	120	136	152	168	184	200	216	232	248
9	9	25	41	57	73	89	105	121	137	153	169	185	201	217	233	249
A	10	26	42	58	74	90	106	122	138	154	170	186	202	218	234	250
B	11	27	43	59	75	91	107	123	139	155	171	187	203	219	235	251
C	12	28	44	60	76	92	108	124	140	156	172	188	204	220	236	252
D	13	39	45	61	77	93	109	125	141	157	173	189	205	221	237	253
E	14	30	46	62	78	94	110	126	142	158	174	190	206	222	238	254
F	15	31	47	63	79	95	111	127	143	159	175	191	207	223	239	255